Constraint-Based Scheduling in Oz

Jörg Würtz, DFKI GmbH, Saarbrücken

Abstract

It is discussed, how scheduling problems can be solved in the concurrent constraint programming language Oz. Oz is the first high-level constraint language, which offers an interface to invent new constraints in an efficient way using C++. Its multi-paradigm features including programmable search are unique in the field of constraint programming. Through the interface, algorithms from Operations Research and related fields can be incorporated. The algorithms can be combined through encapsulation into constraints and can communicate via shared variables. This is exemplified by the integration of new techniques based on edge-finding for job-shop and multi-capacitated scheduling. The viability of Oz as a platform for problem solving is also exemplified by a graphical scheduling workbench. The performance for job-shop problems is comparable to state-of-the-art scheduling tools.

1 Introduction

Since many years, Operations Research (OR) is successfull in solving application problems. Often, tailored algorithms are developped, which tackle the problem under investigation very efficiently. But difficulties may arise, if the tailored algorithm has to be adapted to changes of the problem formulation. The special-purpose program may have to be restated in a time-consuming way.

On the other hand, constraint programming was invented with the aim to combine declarativeness and expressiveness with efficiency [7]. This works well for several application domains like scheduling, configuration, or resource allocation. But for larger and especially hard problems, the used constraint-techniques were not sufficient. Hence, techniques from OR were incorporated into constraint systems, pioneered by `CHIP` [4]. Changes of the problem formulation can be modeled by stating different constraints. But `CHIP` lacks flexibility to allow the user to invent new kinds of constraints.

We propose Oz [11, 12] as a platform to integrate algorithms from OR to achieve an amalgamation of a high-level constraint language with efficient OR techniques. Oz is a concurrent constraint language providing for functional, object-oriented, and constraint programming. The unique advantages of Oz, which can be offered to the OR community are:

- *Expressiveness.* Different language paradigms allow a natural and high-level modelling of the problem. Concurrent constraints provide for rapid testing of different models.
- *Programmable Search.* Besides predefined search strategies like depth-first one solution search or branch & bound, the user can program his own strategies [10]. Search is completely separated from the reduction of the search space achieved by constraint propagation.
- *Modularity.* Through the encapsulation of different algorithms into constraints, they can be combined and interact in one environment. In combination with search and objects, collaborating problem solvers can be programmed.
- *Openness.* Through the use of a C++ interface [9], new constraints can be implemented efficiently by the programmer and used like any Oz procedure.

These advantages are exemplified by a scheduling workbench called *Oz Scheduler* [13]. This workbench makes use of different OR algorithms for propagation and branching, which we have implemented for job-shop scheduling. The user can freely combine different search strategies and constraint algorithms. The performance is comparable to state-of-the-art special-purpose tools for scheduling.

We show the integration of a new technique based on so-called edge-finding techniques [2] into constraints for job-shop and multi-capacitated scheduling. But also techniques coming from linear programming or graph algorithms may be integrated.

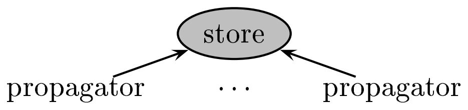
The paper is structured as follows. In Section 2, constraint programming in Oz is introduced. Section 3 explains how scheduling problems can be solved in Oz. In Section 4, the performance is evaluated for job-shop problems. The paper concludes with an overview on related work.

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 [10, 5].

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 non-basic 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. Propagators 'communicate' via shared variables in the store. If a propagator narrows a domain, further propagators may be triggered, which may narrow further domains, triggering propagators and so on. This takes place until a fixed point is reached. Constraint propagation is completely independent from the particular implementation of a propagator. The constraint language serves as the glue to connect all the propagators and their algorithms.

Constraint propagation is usually incomplete (for the sake of efficiency). Hence, to obtain a solution for a set of constraints S, we 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 *choice-point*. The second alternative $S \cup \{\neg C\}$ is solved if the first alternative leads to an inconsistent store (backtracking). 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 interleaved *propagation and distribution steps*. In Oz, distribution strategies like first fail (choose the variable first with the currently smallest domain) but also more elaborated strategies can be programmed by the user.

Additionally, Oz offers programmable search [10]. Besides search for one or all solutions and branch&bound, resource limited search (limited number of time or failures) or local optimization techniques can be programmed; both used in the Oz Scheduler. The search tree for a solution can be visualized by the Oz Explorer.

While propagators can be developed and tested in Oz itself, a more efficient implementation is available through a C++ interface, which allows the efficient usage of destructive datastructures [9]. The interface provides for high-level abstractions to free the programmer from tedious work like triggering propagators etc. For the programmer it is transparent whether a propagator is provided as an Oz procedure or as a builtin through the interface.

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

3 Scheduling in Oz

We consider job-shop problems first. 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. The resources have unary capacity and no preemption is allowed. A job-shop problem is characterized by two kinds of constraints: precedence and resource constraints. The solution of a scheduling problem consists in an assignment of start times to tasks that is consistent with all constraints. Usually, one is also interested in the optimal solution, minimizing the overall length of the schedule. A task can be modeled by its duration and a finite domain for its start time. A precedence constraint like that A with duration $d(A)$ must precede B is stated as

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

A resource constraint states that two tasks A and B on the same resource must not overlap in time, i.e, $A + d(A) \leq B \ \vee \ B + d(B) \leq A$. This can be modeled in Oz by so-called reified constraints:

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

Here, the validity of e.g. $A + d(A) \leq B$ is reflected into the 0/1 valued variable C_1. But the resulting 'local' reasoning is too weak to solve hard problems. Thus, a technique called *edge-finding* was invented in [2]. We explain it in terms of constraint propagation.

Let S be an arbitrary set of tasks to be scheduled on the same resource and $T \in S$. Let S' be S without T. Then, T must be last, if it cannot be scheduled before 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

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

T cannot be scheduled before all tasks in S'. Hence, T must be last and the start time can be narrowed correspondingly, i.e., $T \geq s(S') + d(S')$. Analogous rules hold for the detection that a task must be first. Several approaches differ in the amount of further propagation and the selection of the task sets S to consider [1, 2, 3, 6, 8].

We integrated a kind of edge-finding in a propagator, which bases on an algorithm suggested in [8] for proving lower bounds of job-shop problems. The algorithm (quadratic complexity) computes so-called ascending sets of tasks. For tasks not in these sets, edge-finding rules are applied. By construction, this algorithm avoids some useless edge-finding tests. To improve the algorithm, it is checked for all pairs of tasks, whether one can be scheduled before the other (essentially adopting the reified approach above). One of the advantages of this algorithm is that it can be generalized for multi-capacitated resources (see below).

For distribution (or branching as it is often called in OR), we implemented several strategies from OR and Artificial Intelligence. The best results for proving optimality were obtained by a simplified algorithm of [3], where we avoid the maintenance of cumbersome datastructures. In contrast to

[3], we use the distribution strategy also to add dynamically new propagators (because we can detect tasks, which must be scheduled first or last in a task subset). One surprising observation is that this strategy (providing a very simple form of edge-finding) in combination with a propagator comprising only the propagation of the reified approach above, was sufficient to solve hard scheduling benchmarks (see Section 4).

To find the optimal solution quickly, we used repair and shuffle techniques for local optimization (see e.g. [1]), while the edge-finding propagators are also stated for propagation. For this it was very convenient and inevitable to use programmable search in Oz and high-level abstractions.

Furthermore, we implemented a propagator for multi-capacitated scheduling, where the resources may have a capacity greater than one and the tasks may consume more than one resource unit. To this aim, we generalized our edge-finding algorithm. Essentially, instead of computing the overall duration of a task set, we make use of the amount of resource usage. Let *cap* be the resoure capacity, $u(T)$ the resource usage of a task T, and $a(S) = \sum_{T \in S} d(T) * u(T)$. If

$$(c(S') - s(S')) * cap < a(S') + d(T) * u(T),$$

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

$$(c(S') - s(T)) * cap < a(S') + d(T) * u(T),$$

T cannot be scheduled before all tasks in S'. Let $rest = a(S') - (c(S') - s(S')) * (cap - u(T))$. If $rest > 0$, then T can be narrowed with $T \geq s(S') + \lfloor rest/u(T) \rfloor$.

Moreover, for each task a time interval is computed, where the task occupies the resource in any case (if the latest start time is smaller than the earliest completion time). These intervals are used to disallow time intervals for tasks, which would exceed the available resource capacity. Note that this kind of propagator can also be used for geometrical reasoning (modeling rectangles for furniture layout, for example).

If a problem requires further constraints like that tasks must have a particular distance, this can be accomodated by constraint programming very easily. The propagators using edge-finding can be stated as before, and one only adds the new propagators.

4 Evaluation

In this section we evaluate the performance of Oz for 10x10 job-shop problem instances of [1] in Table 1. For all problems the optimal solution (starting with no information) has to be found and the optimality has to be proved. *Problem* denotes the problem instance in [1], *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. The column *Reified* indicates that reified constraints were used for the resource constraints[1], while *Edge-Finding* indicates that edge-finding was used.

The results on the same problems for ILOG SCHEDULE and Claire (see Section 5) are shown in Table 2 (*BT* denotes the number of backtracks; ILOG used an IBM RS6000 workstation and for Claire a Sparc10/40 MHz was used).

In [1], 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.

This shows that Oz is comparable to state-of-the-art scheduling tools like ILOG SCHEDULE[6].

[1]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.

	Reified				Edge-Finding			
Problem	Fails	CPU	Fails(pr)	CPU(pr)	Fails	CPU	Fails(pr)	CPU(pr)
MT10	5838	169	3983	94	4117	157	2564	81
ABZ5	4295	130	2160	52	3455	138	1597	52
ABZ6	1737	64	239	5	1508	71	200	6
La19	3798	112	1756	40	3331	138	1371	45
La20	4793	129	3247	78	6496	228	1943	57
ORB1	20164	554	16252	399	14242	521	11775	388
ORB2	2813	86	766	17	2421	99	596	19
ORB3	42327	1071	39405	952	34422	1121	28232	850
ORB4	6180	172	1939	45	3722	140	1340	38
ORB5	3987	114	1499	40	3468	138	1155	40

Table 1: Results on 10x10 job-shop problems for the Oz Scheduler

	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 2: Results for ILOG and Claire

5 Related Work

In this section we shortly compare the relation to other constraint systems. `CHIP`[4] also provides for means to solve scheduling problems, but is not open to implement new propagators. It heavily relies on its Prolog implementation. Claire[3] is a language compiling to C++ code, which provides very good scheduling results. Its disadvantage is that it does not provide a rich programming environment. ILOG SCHEDULE[6] is a C++ library dedicated to scheduling. We claim that the exclusive use of C++ makes programming often more complicated than if one uses a high-level language like Oz (offering comparable performance figures).

Remark The documentation of the DFKI Oz system is 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/`.

References

[1] 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.

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

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

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

[5] M. Henz and J. Würtz. Constraint-based time tabling – a case study. *Journal of Applied Artificial Intelligence*, 10(5), October 1996. To appear.

[6] ILOG, URL: `http://www.ilog.com`. ILOG SCHEDULE *2.0, User Manual*, 1995.

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

[8] P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. International Conference on Integer Programming and Combinatorial Optimization, Vancouver, 1996.

[9] T. Müller and J. Würtz. Interfacing propagators with a concurrent constraint language. In *JICSLP96 Post-conference workshop and Compulog Net Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 195–206, 1996. xv.

[10] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A.H. 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.

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

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

[13] J. Würtz. Oz Scheduler: A workbench for scheduling problems. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI'96)*, 1996. To appear.