Interfacing Propagators with a Concurrent Constraint Language

Tobias Müller
Programming Systems Lab
University of the Saarland
Geb. 45, Postf. 15 11 50
66041 Saarbrücken
tmueller@ps.uni-sb.de

Jörg Würtz
Programming Systems Lab
German Research Center for AI
Geb. 45, Postf. 15 11 50
66041 Saarbrücken
wuertz@dfki.uni-sb.de

This paper describes a C++ interface for the concurrent constraint language Oz to implement non-basic constraints as propagators. The programmer benefits from the advantages of a high-level language, like elegant and concise coding, in conjunction with efficiency. For the user it is transparent whether a constraint is implemented by an Oz procedure or through the interface. The interface is completely separated from the underlying Oz implementation. Moreover, it frees the user from tedious tasks like suspending and waking up constraints.

The overall efficiency of the resulting system is comparable to existing finite domain systems. For scheduling applications we demonstrate how algorithms from Operations Research can be incorporated, which allows to obtain results comparable to commercially available systems.

Keywords. Concurrent Constraint Language, Finite Domain Constraints, Efficient Implementation

1. Introduction

A concurrent constraint language provides for a well-suited platform to implement a finite domain constraint system over natural numbers. The concepts of a constraint store and of concurrent computation give the appropriate metaphors needed to understand and implement constraint programs. For the language Oz, we have shown in [SSW94] that the minimal requirements to enable finite domain programming are primitives to constrain a variable to a finite domain and to reflect the current domain of a variable in conjunction with encapsulated search. Desired constraints can be expressed in the language itself in an elegant way.

But finite domain systems must be efficient in order to tackle practical problems, like scheduling, placement or configuration [DVS+88]. Consequently, we decided to reimplement more complex constraints, so-called propagators, which were implemented as Oz procedures by corresponding C++ code. The resulting system provided the required efficiency and still allowed the user to invent new constraints by combining propagators with Oz language constructs like conditionals or disjunctions [ST95, MW96]. Nevertheless, users were not able to implement C++ propagators themselves to meet their individual needs of efficiency.

Hence, we wanted to design an interface to C++ propagators that can easily be used by programmers. The main design goals were as follows:

- It must be *transparent* to the user whether a propagator is implemented by an Oz program or by a C++ propagator implemented through the interface.

- The interface must support a propagator to have properties like *statefulness*, *atomicity* and *efficient data structures*. This is due to the requirements of the algorithms usually employed by propagators. Such algorithms are typically derived from techniques known from mathematics and Operations Research.

- Propagators and the Oz runtime-system are *completely separated*. They are glued together only by the interface abstractions. This enables the implementation of propagators without having access to the code of the Oz runtime-system.

- The programmer is *freed from tedious tasks* like suspending or waking up propagators and caring about computation spaces (see Section 2.1.). This is achieved by providing interface abstractions at an appropriate level.

Since we use C++, some similarities with the ILOG SOLVER are inevitable. The main difference is that in ILOG SOLVER both the problem formulation and the search strategy has to be written in C++. In contrast to ILOG SOLVER, Oz provides for performance-critical constraints the option to implement them more efficient as C++ propagators. The actual program is still an Oz program, i.e. the problem formulation and the embedding of the constraint problem in larger applications is done in a high-level language. Moreover, search itself and the labelling strategies can be programmed in Oz too.

C++ propagators are needed only in performance-critical Oz code segments. Oz is still useful for such code segments as a prototyping language. Code written in Oz can be efficiently debugged and tuned, before it is reimplemented using the interface. Our experiences shows that this approach shortens the development cycle of application programs.

The resulting finite domain system is expressive (it includes reified constraints [HW96], non-linear constraints and several symbolic constraints) and more efficient than Oz 1.0 [ST95] with unseparated propagators, i.e. C++ propagators tightly integrated in the emulator. The viability of our approach is also exemplified by the incorporation of special-purpose propagators for scheduling [Wür96]. The resulting system shows an efficiency comparable to ILOG SOLVER (see Section 5.).

The plan of the paper is as follows. The next section sketches constraint computation in Oz and introduces basic notions and concepts. Section 3. explains in detail the implementation of a propagator, to give the reader an intuition of the expressiveness of the interface. A case study in Section 4. shows how more complex propagators benefit from the interface. Finally, the yielded results are presented and discussed.

2. Computation with Constraints in Oz

2.1. Computation Model

This paper focusses on constraints on finite sets of nonnegative integers, so-called *finite domains*, in the constraint programming language Oz. For a more thorough treatment see [SSW94, HW96, MW96]. For a detailed presentation of the programming model see [Smo95].

A *basic constraint* takes the form $x = n$, $x = y$ or $x \in D$, where x and y are variables, 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* which is *posted on* the variables occurring in the corresponding constraint. It tries to narrow the domains of the variables it is posted on by amplifying the store with basic constraints. This narrowing is called *constraint propagation*. A propagator P amplifies the store S by writing a basic constraint ϕ to it, if $P \wedge S$ entails ϕ but S on its own does not. If P ceases to exist, it is either entailed by the store S, or $P \wedge S$ is unsatisfiable. Note that the amount of propagation depends on the operational semantics of the propagator. For example, the equation $X + Y = Z$ can be modelled by the conjunction of the two propagators $X + Y \leq Z$ and $X + Y \geq Z$ instead by only one propagator.

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 1. We say that the propagator $X + Y < Z$ *constrains* the variables X, Y and Z.

In general, computation in Oz is built around the notion of a *computation space*. A computation space consists of the *constraint store* and *tasks* connected to the store. One kind of tasks are propagators. Because tasks themselves (like disjunctions) may host computation spaces, a tree of spaces results (see [Smo95]). Furthermore, encapsulated search in Oz employs local computation spaces too.

2.2. Spaces and Propagators

As indicated above, a propagator is a computational agent that is associated with a certain space, called its *home space* (for details see [MSS95]). There are three kinds of data flow between a space and a propagator.

1. *Reading and writing basic constraints.* A propagator reads the basic constraints of the variables it is posted on. In the process of constraint propagation it writes basic constraints to the store.

2. *Status of propagation.* The propagator's home space derives its status from the statuses of the propagators it hosts. A single failed propagator causes the whole computation space to fail. The existence of a single propagator prevents a space from being entailed. A necessary condition for a space to be entailed is that all propagators cease to exist.

3. *Synchronisation.* A propagator's home space notifies the propagator when the store has been amplified in a way the propagator is waiting for. For example, many propagators will be notified only in case the bounds of a domain have been narrowed.

Figure 1 depicts the data flow between a propagator and its home space.

As a first approximation, a propagator can be regarded as a stateful long-lived computational entity which is waiting on a synchronisation message, is able to read and write basic constraints, and signals the status of propagation.

2.3. Adding Propagators to Oz

The computation model sketched above is realized by the Oz runtime system, which is implemented by an abstract machine [MSS95], called the *emulator*.

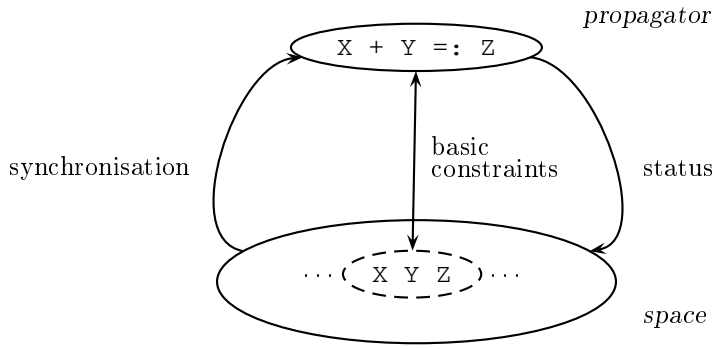

Figure 1: Data flow between space and propagator

Implementation of Propagators. The creation of a propagator is triggered by the execution of an Oz program (see Section 3.2.). When a propagator is created a reference to it is added to the suspension lists of the variables it is posted on; we say, a propagator is *suspending* on these variables. This ensures the run of the propagator whenever these variables are narrowed. The entries in the suspension lists are used to synchronise a propagator's execution on narrowed domains.

A propagator has direct access to the constraint store through its arguments which are stored in its state. Consequently, reading and writing basic constraints is done by the propagator itself. That causes the propagator to be responsible for scanning suspension lists of variables it is posted on. An appropriate interface abstraction takes care of that tedious matter. In case the propagator amplifies the constraint store, i.e. writes basic constraints to the store, it notifies the emulator what other propagators have to be synchronised (that means *woken up*). A propagator run stops if the constraints in the store cannot be further amplified by the propagator or are inconsistent with the propagator.

The value returned by a propagator to the emulator is either `SLEEP`, `ENTAILED` or `FAILED`. The value `SLEEP` indicates that the propagator continues to exist and the propagator is consistent with the constraint store, but for now, cannot further amplify the store. The value `FAILED` is returned if the propagator is inconsistent with the store and consequently, has to cease to exist. A propagator which consistently amplified the store with respect to its semantics to a maximal extend ceases to exist and returns `ENTAILED`. The emulator uses the return value to compute the status of the propagator's home space.

A Propagator's Execution State. From the emulator's perspective, a propagator is a stateful long-lived computational agent, which exists in different execution states and has to be provided with resources like computation time and heap memory.

The emulator's scheduler switches a propagator between the execution states and inevitably for a concurrent computation model, provides computation time in a fair way. That means that every propagator that is `runnable` will eventually be run.

After creation, a propagator has one of five different states (namely `running`, `runnable`, `sleeping`, `entailed`, and `failed`) which are controlled by the emulator's scheduler (see Figure 2). When a propagator is created, its state is immediately set `running` and the scheduler allocates a time slot for its first run. After every run, the emulator evaluates the propagator's return value

which either causes the propagator to be terminated (ENTAILED resp. FAILED) or causes it to be kept alive (SLEEP).

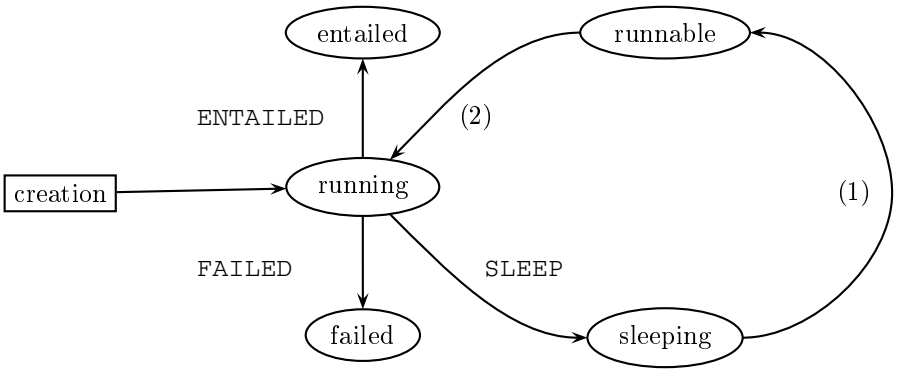

Figure 2: Execution states of a propagator

Whenever a propagator returns SLEEP and its state is switched to `sleeping`, the emulator keeps the propagator suspending on its constrained variables. A propagator is woken up when variables it is posted on are further constrained. This is indicated by (1) in Figure 2. The propagator's state is then switched to `runnable`. Now, the scheduler takes care of the propagator and will schedule it later on (the transition (2) from `runnable` to `running` is subject to the scheduler's policy and will be not discussed here). If the propagator fails its execution state is set to `failed` and the current computation space will be discarded by the emulator. If the propagator can decide (according to its operational semantics), that the corresponding constraint is entailed, it returns ENTAILED and its execution state is set to `entailed`. As optimisation the emulator removes all suspension entries from variables the propagator was previously been posted on.

3. The Constraint Propagator Interface

A major design objective of the finite domain system of Oz was to be as high-level as possible and as low-level as necessary. Therefore, we provide for Oz a full-fledged ready-to-use finite domain system. But certain problems require tailored algorithms, which can often better be implemented in an imperative language rather than in Oz. Thus, a set of C++ abstractions is provided, to connect propagators with the emulator; in this way, the implementation of propagators has been completely separated from the rest of the emulator.

To prove the feasibility of our approach we reimplemented the finite domain constraint library of Oz using the constraint propagator interface. In fact, this library is dynamically loaded by the emulator when launching the system and mapped to the corresponding Oz abstractions. There was no performance penalty for using the propagation interface.

Another design objective was, of course, the compatibility with Oz; the implementation of a propagator through the interface should be transparent. The major difficulties here are that variables a propagator is posted on, may belong to different computation spaces. Furthermore, equality between variables imposed at runtime must be considered (note that in Oz, equality is a basic constraint of the store). The difficulty of local computation spaces is overcome by hiding it

away by sufficiently powerful interface abstractions. Equality between variables can be detected by appropriate interface functions.

Every relevant entity of the emulator, like for example a variable in a store constrained with a finite domain, is represented in the interface by a corresponding C++ class. The member functions of these classes provide the required functionality to implement finite domain propagators.

3.1. Building a Propagator

This section explains by means of an example the constraint propagator interface of Oz. We implement the propagator for the domain-consistent constraint $X + Y = Z$. For the sake of clarity we use a rather naive implementation here.

The Interface Class `OZ_Propagator`. The emulator requires a uniform way to refer to all instances of propagators. This is realized in the interface by the virtual base class `OZ_Propagator`, which is the base class for all propagators. Consequently, a propagator is implemented as a C++ object. The class `OZ_Propagator` defines the basic functionality required by the emulator. The programmer has to replace the pure virtual member functions to build a propagator.

```
enum OZ_Return {ENTAILED, FAILED, SLEEP};

class OZ_Propagator {
public:
  OZ_Propagator(void);
  virtual ~OZ_Propagator(void);
  static void * operator new(size_t);
  static void operator delete(void *, size_t);
  virtual OZ_Return run(void) = 0;
};
```

The operators **`new`** and **`delete`** are necessary to make propagators compatible with the emulator's memory management.

The central member function is `run`. It is responsible for the actual constraint propagation and is called by the emulator when the propagator's execution states is switched to **`running`**. The implementation of a propagator requires first the definition of a new class for the propagator.

```
class PlusPropagator : public OZ_Propagator {
private:
  OZ_Term _x, _y, _z;
public:
  PlusPropagator(OZ_Term a, OZ_Term b, OZ_Term c)
   : _x(a), _y(b), _z(c) {}
  virtual OZ_Return run(void);
};
```

The example propagator stores in its state references to variables it is posted on (namely `_x`, `_y` and `_z`). A value of type `OZ_Term` is used to refer to a variable on the heap. The constructor of the class `PlusPropagator` simply initialises the data members. We will see later where the constructor is used. We first describe two further classes, which are used in the function.

The Member Function `OZ_Propagator::run`. The algorithm for the example propagator is straightforward. The domains of the variables are always rebuilt on each invocation. Therefore

auxiliary domains for each variable are introduced which are initially empty. For all values of the domains of X and Y it is checked if there is a consistent value in the domain of Z. If so, the values are added to corresponding auxiliary domains. Finally, the domains of the variables are constrained with the corresponding auxiliary domains. Consequently, the core of the program code consists of two nested loops iterating over all values of the domains of X and Y.

```
OZ_Return PlusPropagator::run(void)
{
  OZ_FDIntVar x(_x), y(_y), z(_z);
  OZ_FiniteDomain x_aux(fd_empty), y_aux(fd_empty),
                  z_aux(fd_empty);

  for (int i=x->getMinElem(); i!=-1; i=x->getNextLargerEl(i))
    for (int j=y->getMinElem(); j!=-1; j=y->getNextLargerEl(j))
      if (z->isIn(i + j)) {
        x_aux += i;
        y_aux += j;
        z_aux += (i + j);
      }
  FailOnEmpty(*x &= x_aux);
  FailOnEmpty(*y &= y_aux);
  FailOnEmpty(*z &= z_aux);
  return (x.leave()|y.leave()|z.leave()) ? SLEEP : ENTAILED;

failure:
  x.fail();
  y.fail();
  z.fail();
  return FAILED;
}
```

The Interface Class `OZ_FDIntVar`**.** A propagator needs direct access to the constrained variables it is posted on. The interface class `OZ_FDIntVar` member functions access variables in the store directly. The constructor dereferences the variable in the store and stores the dereferenced information in the state of the newly created object. The operators `*` and `->` are overloaded to provide direct access to the representation of the finite domain of a variable in the store resp. to invoke member functions of the class `OZ_FiniteDomain` (see below). The member function `leave` has to be called when the propagator is left. If the variable's domain has been constrained by the propagator, it causes the scheduler to switch all propagators waiting for further constraints on that variable to come to `runnable`. The return value of `leave` is 0 if the domain became a singleton, otherwise 1. This information is used to decide whether a propagator is entailed or not. The member function `fail` is to be called if the propagator encounters an empty domain and does some cleanups.

The Interface Class `OZ_FiniteDomain`**.** The basic finite domain constraint on a variable is represented by an object of the class `OZ_FiniteDomain`. Modifying their value is immediately visible in the constraint store. The operator `+=` adds a value to a domain. The operator `&=` intersects two domains, modifies the domain on the left hand side and returns the size of the intersected domain. The member function `getMinElem` returns the smallest value of the domain and `getNextLargerEl` returns the smallest value of the domain larger than `i`. Testing whether a

value is contained in a domain or not can be done by the member function `isIn`.

The Implementation. The implementation of the constraint $X + Y = Z$ proceeds as follows. First the variables in the store are dereferenced and stored in the local C++ variables `x`, `y` and `z`. The corresponding auxiliary domains are held in the variables `x_aux`, `y_aux` and `z_aux`, which are initialised to empty domains. Two nested for-loops enumerate all possible pairs (v_x, v_y) of values of the domains of X and Y. Each loop starts from the smallest value of a domain and proceeds till -1 is returned, indicating, that there is no larger value. If there is a value v_z in the domain of Z satisfying the relation $v_x + v_y = v_z$ then these values are added to the appropriate auxiliary domains. After completing the nested loops, the domains of the variables are constrained by intersecting them with the auxiliary domains. The macro `FailOnEmpty` branches to the label `failure` if its argument results in the value 0. By this means, constraining the domain of a variable to an empty domain would cause the execution to branch to label `failure` and eventually return `FAILED` to the emulator. The return value of the member function `leave` of class `OZ_FDIntVar` is used to decide whether the propagator returns `SLEEP` or `ENTAILED`. The return value `ENTAILED` indicates entailment and is returned if all variable's domains are singletons, i.e. the calls of `leave` for all three variables return 0. Otherwise, `SLEEP` is returned and the propagator is reinvoked when at least one of its variables is constrained again.

3.2. Creating a Propagator

Before a propagator can be created and introduced to the emulator, its variables must be sufficiently constrained, e.g., for the example the variables must be constrained to finite domains. In case only a subset of variables is sufficiently constrained, the computation will suspend and resume again when more constraints become available. This is checked in a separate C function, which is directly called from the emulator, as consequence of applying an Oz procedure connected with this C function. Further, when a propagator is posted on a variable, it has to be determined which changes to the domain wake up the propagator again. The alternatives are to wake up a propagator if the variable's domain becomes a singleton, the bounds are narrowed or some value is removed from the domain.

The macros `OZ_C_proc_begin` and `OZ_C_proc_end` are provided to allow the implementation of C functions which are compliant with the calling conventions of Oz's emulator.

```
OZ_C_proc_begin(plus, 3)
{
  OZ_Expect pe;

  EXPECT(pe, 0, expectIntVarAny);
  EXPECT(pe, 1, expectIntVarAny);
  EXPECT(pe, 2, expectIntVarAny);

  return pe.spawn(new PlusPropagator(OZ_args[0], OZ_args[1],
                                     OZ_args[2]));
}
OZ_C_proc_end
```

The interface class `OZ_Expect` provides the required functionality to fulfill the above mentioned task. For the sake of program code conciseness the macro `EXPECT` is supplied by the interface. It ensures that incompatible constraints are rejected and insufficient constraints cause the execution to be suspended until more constraints become known. An object of class `OZ_Expect` collects in its state all variables the propagator is supposed to be posted on. The member function

`expectIntVarAny` of class `OZ_Expect` expects a variable already constrained to a finite domain. If a variable is sufficiently constrained, it is stored in the state of the object `pe`. A side-effect of a member function of the group `expectIntVar` is that it determines which kind of pruning of the domain will wake up the propagator. For example, the member function `expectIntVarSingl` instead of `expectIntVarAny` would cause the propagator to be woken up, if the domain of the appropriate variable becomes a singleton.

Finally, the actual propagator is created by calling its constructor via the application of the **`new`** operator. The reference to the newly created propagator is passed as argument to `spawn`, a member function of `OZ_Expect`, which executes the `run` method and introduces the propagator to the emulator.

4. Implementing Complex Propagators

In this section we outline how to implement a more complex propagator in the field of scheduling. In scheduling problems several tasks compete for shared resources. If only one task can be served by a resource at once, the constraint has to be stated that no two tasks on this resource overlap in time. This can be expressed by stating for all task pairs the disjunction

$$s(X_1) + d(X_1) \leq s(X_2) \quad \vee \quad s(X_2) + d(X_2) \leq s(X_1)$$

where $s(X_i)$ and $d(X_i)$ denote the start time and the duration of a task, respectively. If n tasks are to be scheduled on the same resource, these constraints can be stated by imposing $n(n-1)/2$ propagators modelling the described disjunctions.

An alternative which saves memory is to model these constraints by only one propagator, called disjunctive propagator in the sequel. To avoid useless computation (e.g., in case one disjunction becomes valid), the propagator maintains a state that keeps track of only those disjunctions that have already been committed to a clause.

The disjunctive propagator will be posted on two tuples containing start times and durations of tasks scheduled on the same resource. The propagator suspends only on the start times, whereas the durations are stored in the state. Furthermore, the propagator's state holds pairs of integers, where a pair (i, j) denotes that the corresponding tasks for i and j must not overlap.

The member function `run` of the disjunctive propagator will check for each pair if the corresponding disjunction can be discarded. If one clause of a disjunction is entailed by the current constraint store, e.g. $\max(s(X_1)) + d(X_1) \leq \min(s(X_2))$, the corresponding pair has not to be considered anymore in further invocations of the propagator and can be destructively deleted from the stored pair list. Here, *min* and *max* denote the functions returning the current minimum and maximum of a domain, respectively. If one clause is disentailed (e.g. $\min(s(X_1)) + d(X_1) > \max(s(X_2))$, the remaining clause must be spawned as a propagator and the corresponding pair can be deleted from the list. To this aim a further primitive is provided by the interface to spawn another propagator from a running propagator (in our case the disjunctive propagator). On the other hand, the disjunctive propagator must loop over the pairs until no pair can be discarded anymore (since the propagation behaviour of several disjunctions is modelled). Since the execution of a propagator is atomic, the spawned propagators cannot run until the disjunctive propagator has finished. Hence, to get the immediate effect spawning a propagator of a clause is modelled inside the disjunctive propagator by narrowing the domain of the variables appropriately (e.g. if the first clause fails, the constraints $Y \leq \max(s(X_1)) - d(X_2)$ and $X_1 \geq \min(s(X_2)) + d(X_2)$ are added to the constraint store immediately).

As optimisation, the disjunctive propagator unposts itself from tasks that do not occur anymore, which includes removing the suspensions to the propagator from the tasks's suspension lists.

Because more complex propagators are usually more expensive in terms of computation time, the interface provides also means to delay such propagators by assigning them a lower execution priority.

5. Evaluation

The performance of the interface is evaluated on a set of standard benchmarks in Table 1. We compare our work with `clp(fd)` [DC93] and ECL^{i}PS^e [ECR95]. The *queens* problem is the usual one. The problems *alpha* and *donald* are crypto-arithmetic puzzles. Finding a solution for a set of 10 and 20 equations is the task of the problems *equation10* and *equation20*, respectively. The annotations indicate the labelling strategies; either first-fail or naive. Note for these benchmarks `clp(FD)` is faster than Oz due to the fact that Oz uses copy-based search, has an emulator-based runtime system and implements a far more complex computation model.

Problem	Oz (secs)	`clp(fd)` (secs)	`clp(fd)` / Oz	ECL^{i}PS^e (secs)	ECL^{i}PS^e / Oz
16-queens, naive	18.47	2.7	0.146	15.48	0.838
30-queens, ff	2.17	0.49	0.225	3.22	1.484
alpha, naive	41.5	16.69	0.402	256.5	6.181
alpha, ff	0.34	0.24	0.705	1.35	3.970
donald, naive	19.48	6.23	0.319	21.2	1.088
equation20, naive	0.43	0.18	0.418	1.10	2.558
equation10, naive	0.28	0.12	0.428	0.53	1.893

Table 1: Comparing different finite domain systems on a SPARC ELC

The propagator presented in Section 4. in conjunction with an improved labelling strategy based on [CL94] was used to tackle job-shop problems[1], which were supposed to be particularly hard for a long period of time.

The obtained results are shown in Table 2. The first two columns give the number of failures and the time taken to find the optimal solution from scratch and to prove the optimality. The last two columns indicate the numbers of failures and the time taken for proving the optimality only. The runtimes are comparable to the results obtained with ILOG SCHEDULE in [BP95] (on an IBM RS6000). More details on scheduling in Oz can be found in [Wür96].

Acknowledgements

The authors would like to thank Martin Müller and Peter Van Roy for their invaluable comments on a draft version of this paper.

[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.

Problem	Fails	CPU time (secs)	Fails (pr)	CPU time (pr) (secs)
MT10	5 812	278	3 982	150
ABZ5	4 371	219	2 159	77
ABZ6	1 712	109	238	7
La19	3 709	182	1 755	62
La20	4 849	184	3 246	117
ORB1	19 450	885	16 251	623
ORB2	2 813	148	765	27
ORB3	47 446	1 855	39 405	1 141
ORB4	6 052	287	1 938	68
ORB5	3 971	194	1 498	57

Table 2: Results of 10x10 job-shop problems on a SPARC 20

References

[BP95] P. Baptiste and C. Le 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.

[CL94] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming*, pages 369–383, 1994.

[DC93] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *Proceedings of the International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. MIT Press.

[DVS+88] 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.

[ECR95] ECRC. *ECL^{i}PS^e, User Manual Version 3.5*, December 1995.

[HW96] M. Henz and J. Würtz. Using oz for college timetabling. In E.K. 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.

[MSS95] M. Mehl, R. Scheidhauer, and C. Schulte. An Abstract Machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, Lecture Notes in Computer Science, Utrecht, The Netherlands, 20–22September 1995. Springer Verlag.

[MW96] 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.

[Smo95] 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.

[SSW94] 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.

[ST95] G. Smolka and R. Treinen, editors. *DFKI Oz Documentation Series.* Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1995.

[Wür96] J. Würtz. Oz Scheduler: A Workbench for Scheduling Problems. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI'96)*, 1996. To appear.