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

Abstract

This paper deals with an interface for the concurrent constraint language Oz to implement arbitrary constraints in an efficient manner. The programmer has all the advantages of a high-level language in combination 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 abstract machine of Oz. Moreover, it frees the user from tedious tasks like suspending and waking of constraints.

As an implementation language we use C++. The overall efficiency of the resulting system is comparable to existing finite domain systems. For scheduling applications we show that algorithms from Operations Research can be incorporated, which allows to obtain results comparable to commercially available systems.

Keywords concurrent constraint language, finite domain constraints, propagator interface.

1 Introduction

Concurrent constraint languages provide the ideal means to implement a finite domain constraint system over natural numbers. The inherent notions of a constraint store and of concurrent computation give the right metaphors needed to understand and develop constraint programs. For the language Oz, we have shown in [SSW94] that finite domain programming is enabled by adding primitives to reflect the current domain of variables and encapsulated search. Arbitrary constraints can be expressed in the language itself.

But finite domain systems must be efficient in order to tackle the problems, which made this field so remarkable, like scheduling, placement or configuration [DVS+88]. Hence, we decided to encapsulate several constraints into so-called propagators written in C++, i.e., constraints as procedures were replaced by constraints as C-functions. The resulting system was rather efficient (faster than ECL^{i}PS^e for standard benchmarks) and still allowed the user to invent new propagators by combining propagators with language constructs like conditionals or disjunctions [ST95, MJ]. Nevertheless, users could implement new constraints efficiently only in a cumbersome way (by compiling the whole abstract machine).

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

- It must be *transparent* whether a propagator is implemented by an Oz program or the interface.
- The interface implementation language must allow to incorporate *state*, *atomicity* and *efficient data structures*. This is necessary because algorithms adopted from Operations Research often require that. Furthermore, it should provide the *right level of abstraction* to ease programming (object-oriented programs).
- The abstract machine of Oz should be *completely separated* from the interface.
- The programmer should be *freed from tedious tasks* like suspending and waking of propagators.

The language we have chosen for the interface implementation is C++. Therefore, some similarities to the Ilog Solver are inevitable. The main difference is that in Ilog Solver also the problem formulation and the search strategy has to be written in C++. In Oz, only the propagators may be implemented through the interface. For the problem formulation and the embedding of the constraint problem in larger applications one can use all the benefits of a high level language. Propagators may first be written in Oz, debugged and tuned, before they are reimplemented by the interface. Moreover, search itself and the labelling strategies can be programmed in Oz itself.

This paper deals with the implementation of the C++ interface to propagators for the language Oz. The resulting finite domain system is expressive (including reified constraints [HW95], non-linear constraints and several symbolic constraints) and even more efficient than Oz 1.0 [ST95]. The viability of our approach is also exemplified by the incorporation of special-purpose propagators for scheduling. The obtained results are comparable to Ilog Solver.

The plan of the paper is as following. The next section sketches constraint computation in Oz and introduces basic notions and concepts. The following section 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 real-world problems can be solved. Finally, the yielded results are presented and discussed.

2 Computation with Constraints in Oz

2.1 Programming Model

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 [SSW94, HW95]. 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 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 by amplifying the store with basic constraints. This narrowing is called *constraint propagation*. A propagator P amplifies the store S by writing a basic constraints ϕ to it, if $P \wedge S$ entails ϕ but not S on its own. 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 one. 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 the propagators. Because tasks itself (like disjunctions) may host computation spaces, a tree of spaces results (see [Smo95]). Furthermore, encapsulated search in Oz heavily depends on local computation spaces.

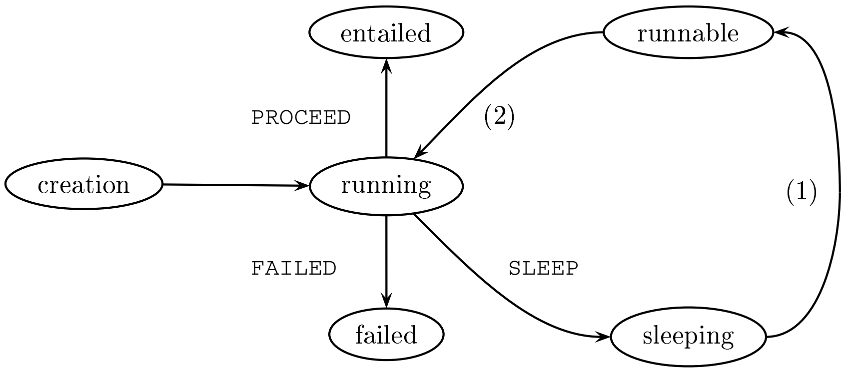

Figure 1: Life cycle of a propagator

2.2 Implementation of Propagators

The sketched computation model of Section 2.1 is implemented by an abstract machine [MSS95], called the *emulator*. A propagator is realized as a C++ object narrowing the variables it is constraining and returning values to the emulator, which indicate the state of the propagator. At each variable is attached the list of suspending tasks (often only propagators), which are constraining this variable. If the variable is changed, propagators suspending on it are woken up by the emulator. If narrowing takes place in the course of computation in a propagator, the propagator itself is waking up suspending tasks. The value returned by a propagator indicates the emulator whether a propagator detected an inconsistency, whether it detects entailment or whether it continues to exist.

For the emulator a propagator is a black-box. Inevitable for a concurrent language, it has to be provided in a fair way with computation time. This is done by the scheduler of the emulator.[1]

A Propagator's Life-cycle A propagator exists in three different states (`running`, `runnable` and `sleeping`), which are controlled by the scheduler of the emulator. When a propagator is created, its state is immediately set `running`, which 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 (`PROCEED` resp. `FAILED`) or causes it to be kept alive (`SLEEP`).

[1]Hence, a strategy used by systems like `clp(FD)` or ECL^{i}PS^e which do constraint propagation until quiescence without interruption, is not applicable to Oz. Here, we have to check, whether other computation has to be scheduled. Our implementation shows, that the performance loss is about 10 % (depending on the problem).

A propagator falls asleep if it cannot amplify the constraint store further but is not entailed yet, according to its operational semantics. In this case, the 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 constraining are changed (see Section 3). This is indicated by (1) in Figure 1. 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 scheduling policy and will be not discussed here). If the propagator fails, 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 `PROCEED` and ceases to exist. The emulator notices variables, that the propagator does not suspend on them anymore.

3 Opening Oz's Emulator to the Programmer: The Constraint Propagator Interface

A major design objective of Oz's finite domain constraint system 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 be better implemented in an imperative language rather than in Oz (like algorithms from Operations Research). That is why we opened up the system to the programmer by completely separating the emulator from the implementation of the propagators. Considering our design goals of Section 1, C++ was the language of choice for the interface. Further, C++ has proved to be suitable for this sort applications [PL95].

To prove the feasibility of our approach we implemented the whole propagator library of Oz using the constraint propagator interface. In fact this library is dynamically loaded to the emulator, when launching the system and mapped to Oz abstractions.

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 local computation spaces and equality between variables imposed at runtime. The first difficulty is overcome by hiding it away by sufficiently powerful interface abstractions. Equality between variables can be detected by appropriate interface functions.

Every entity of the emulator which needs to be reflected via the interface is represented by a 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 the means of an example the constraint propagator interface of Oz. We implement the propagator for the constraint $X + Y = Z$.

The Interface class `OZ_Propagator`. The emulator requires a uniform way to refer to all instances of propagators. This is realized by the interface virtual base class `OZ_Propagator`, which is the base class for all propagators. This implies that a propagator is implemenented 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.

```
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 destructor is made virtual to force the destructors of the derived classes to be virtual too. This is for example important when a propagator is disposed by the emulator. The operators `new` and `delete` are necessary to make propagators compatible with the emulator's memory management.

The central member function is `virtual OZ_Return run(void)`. It is responsible for constraint propagation and is called by the emulator when the propagator's state is switched to `running`. The return type `OZ_Return` can be one of the following values: PROCEED,FAILED or SLEEP. 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`). References to terms, where variables belong to, are of type `OZ_Term`. The constructor of the class `PlusPropagator` simply initialises the data members. We will see later where the constructor is used.

The Member Function `OZ_Propagator::run`**.** The algorithm for the example propagator is straightforward. The domains of the variables are always rebuilt from scratch. 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 : PROCEED;

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` provides member functions to 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 the store variable resp. to invoke member functions of the class `OZ_FiniteDomain` (see below). The member function `int leave(void)` 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 the that variable to come to `runnable`. The return value is 0 if the domain became a singleton, otherwise 1. This information can be used to decide whether a propagator is entailed or not. The member function `void fail(void)` is to be called if the propagator encounters an empty domain.

The interface class `OZ_FiniteDomain`. The finite domain constraint of 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 `int getMinElem(void)` returns the smallest value of the domain and `int getNextLargerEl(int i)` 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 `int isIn(int i)`.

The implementation. The implementation of the constraint $X + Y = Z$ proceeds as follows. First, the variables in 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 expression produces 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 `PROCEED`. The return value `PROCEED` 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 constraint the variables must be constrained to finite domains. In case only deficient constraints are available, the computation will be interrupted and resumed if 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 an arbitrary 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 suspend 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. Collecting variables is a side-effect of checking for sufficient constraints. 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`. Another 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` 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 `OZ_Return spawn(OZ_Propagator *)`, a member function of `OZ_Expect`. It executes the `run`-method and if the value `SLEEP` is returned, the propagator is handed over to the emulator.

4 Implementing Complex Propagators

In this section we show 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

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

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

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. one disjunction becomes valid), the propagator maintains a state.

The disjunctive propagator will be spawned with two tuples containing the start times and durations of the tasks scheduled on the same resource. The propagator suspend only on the start times, whereas the durations are stored in the state. Furthermore, the spawning built-in stores pairs of integers, where a pair (i, j) denotes that the corresponding tasks for i and j must not overlap.

The run-method 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 pair has not to be considered in further calls of the propagator and can be 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 imposed as a propagator. To this aim a further primitive is provided by the interface to spawn another propagator from a running propagator. On the other hand, the loop over the pairs must be run until no more pair can be discarded (since the propagation behaviour of several propagators is modelled). Hence, the effect of imposing a propagator of a clause is modelled 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.

If a task does not occur anymore in a pair, it can be deleted from the call argument. Additionally, the interface provides for means to delete the suspension to the disjunctive propagator from the task's suspension list. Hence, only variables, which can contribute to propagation can trigger the propagator.

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 priority than default.

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 n equations is the task of the problem equation$n$. The annotations indicate the labelling strategies; either first-fail or naive.

Problem	Oz (secs)	`clp(fd)` (secs)	$\frac{\texttt{clp(fd)}}{\text{Oz}}$	ECL^{i}PS^e (secs)	$\frac{\text{ECL}^i\text{PS}^e}{\text{Oz}}$
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 a labelling strategy proposed by [CL94] was used to tackle job-problems[2], 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.

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

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

[2] 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.

References

[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^iPS^e, User Manual Version 3.5*, December 1995.

[HW95] M. Henz and J. Würtz. Using oz for college timetabling. In *International Conference on the Practice and Theory of Automated Time Tabling*, pages 283–296. Springer Verlag,, August/September 1995.

[MJ] 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-96-05.*

[MSS95] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95*, Lecture Notes in Computer Science, vol. 982, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[PL95] Jean-Froncois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Logic Programming - Proceedings of the 1995 International Symposium*, number 02142, pages 513–527. The MIT Press, December 1995.

[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.