A Toolkit for Constraint-based Inference Engines

Tee Yong Chew, Martin Henz, and Ka Boon Ng

National University of Singapore, Singapore 117543,
{henz,ngkaboon}@comp.nus.edu.sg,
WWW home page: http://www.comp.nus.edu.sg/

Abstract. Solutions to combinatorial search problems can benefit from custom-made constraint-based inference engines that go beyond depth-first search. Several constraint programming systems support the programming of such inference engines through programming abstractions. For example, the Mozart system for Oz comes with several engines, extended in dimensions such as interaction, visualization, and optimization. However, so far such extensions are monolithic in their software design, not catering for systematic reuse of components.
We present an object-oriented modular architecture for building inference engines that achieves high reusability and supports rapid prototyping of search algorithms and their extensions. For the sake of clarity, we present the architecture in the setting of a C++ constraint programming library. The SearchToolKit, a search library for Oz based on the presented architecture, provides evidence for the practicality of the design.

1 Introduction

Finite domain constraint programming (CP(FD)) grew out of research in logic programming. The first programming systems for CP(FD), including earlier versions of Ilog Solver [Pug94], had Prolog's depth-first search built-in as their only inference engine.

The language CLAIRE [CL96] and the most recent version of Ilog Solver [ILO99a] (see discussion in [LP99]) support the programming of backtracking-based inference engines. Oz [Smo95] allows the programming of copying-based engines [Sch97b] through a a built-in data type called space.

Mozart [Moz99], the most recent implementation of Oz, provides engines for depth-first search (with or without branch-and-bound), limited discrepancy search [HG95] and parallel search. The Oz Explorer [Sch97a] is a graphical tool that allows to visualize and to interactively explore search trees.

The programming of new tree search algorithms can be a complex task. A true software engineering problem arises when several design dimensions need to be considered. Such dimensions include for example user interaction with the inference engine during search, visualization of the search tree, and optimization via a constraint-based cost function (branch-and-bound).

The engines included in the Mozart search libraries are monolithic in that they combine a basic inference engine with a fixed set of extensions. For example, the class `Search.object` combines depth-first search with interaction on the level of solutions, memory utilization based on recomputation and branch-and-bound optimization. The Oz Explorer [Sch97a] adds to these visualization and interaction on the level of nodes. Due to the monolithic design, the effort spent on implementing extensions of previous engines to support for example visualization of search trees is virtually lost for programming a new search algorithm. In practice, the monolithic structure discourages experimenting with application-specific engines, because it becomes a major effort to implement such a new engine and equip it with useful facilities like visualization of search trees.

The question that is addressed in this work is how to overcome this situation. The answer is to modularize inference engines in such a way that the basic engines and the various design dimensions can be implemented separately, using well-defined interfaces between them. We shall present a design that allows to plug together search algorithms with modules for optimization, interaction, visualization, etc. If a search algorithm is constructed following the outlined guidelines, it becomes trivial to equip it with branch-and-bound optimization, node-level tracing, Oz-Explorer-like branch-and-bound optimization or visualization of the search tree, and other features. Modules for individual dimensions can be developed independently and employed by all search algorithms.

2 Scripts, Stores, Engines

Finite domain problems are problems of assigning integer values to variables such that all given constraints are satisfied. A simple example is

$$x \in \{5, \ldots, 10\}, y \in \{1, \ldots, 7\}, x < y, x + y = 11$$

In finite domain constraint programming, the domains of variables, i.e. the set of its possible values, are kept in a store. Initially, the domains are unrestricted. Simple constraints such as $x \in \{5, \ldots, 10\}$ and $y \in \{1, \ldots, 7\}$ can be directly entered in the store by restricting the domains of the variables in the store. More complex constraints such as $x < y$ and $x + y = 11$ are operationalized using propagators, which are connected to the store. Propagators observe the domains of their variables and strengthen the store according to a propagation algorithm. For example, the propagator for $x < y$ may eliminate from the domain of x all values that are greater than or equal to the biggest value in the domain of y.

Systems that support the programming of inference engines provide abstractions that represent the store. Ilog Solver [ILO99a] represents stores by C++ objects of class `IlcManager`, and Oz represents stores by data of a built-in datatype `Space`. To allow a concise presentation of our results, we present them in the setting of a C++ constraint programming library, assuming the reader to be familiar with C++ syntax. Stores are represented by objects of class `store`. The member function

```
var store::newvar(int low, int high);
```

Program 1 Representing Variables and Propagators

```
void main(int argc, char * argv[]) {
   store * s = new store();
   var x = s->newvar(5,10);
   var y = s->newvar(1,7);
   LessThan(s,x,y);
   Sum(s,x,y,11);
   ...
}
```

introduces a new variable in a given store and returns an identifier by which the variable can be referred to. Propagators are represented by instances of propagator classes, whose creation leads to installation of a propagator in a store. For example, the following function installs a less-than propagator in a given store `s`.

```
void LessThan(store * s, var left, var right);
```

A store that represents the above constraints can be created as shown in Program 1.

The member function `tell` of stores is used by propagators and allows to narrow the domain d_1 of a given variable such that it contains only values from the domain d_2 passed to `tell`.

```
store::tell(var v, int lo, int hi);
```

If the intersection of d_1 and d_2 is empty, a failure occurs. Such failures are crucial for constraint programming, since they allow to prune the search tree. As a generic way to indicate failure to search algorithms, failing `tell` operations raise the C++ exception `Failure()`. The design and implementation algorithms for propagators is well developed in systems such as Ilog Solver [ILO99a] and the Oz Constraint Programming Interface [Moz99]. In this work, we concentrate on the design of tree search algorithms.

3 Search Trees

Usually propagation alone does not suffice to solve constraint problems. Thus, after exhaustive propagation, a distribution step is done, i.e. a constraint c is chosen and search is performed recursively with two stores to which c and, respectively, $\neg c$ are added. An algorithm that systematically generates suitable constraints for distribution is called a distributor, and the constraints c and $\neg c$ define a choice point. Distributors are represented in our library by extension of an abstract class `choice`.

```
class choice {
public:
```

Program 2 Naive Variable Enumeration

```
class naive : public choice {
private:
    vector<var> vars; // variables to be enumerated
    int idx;          // index of variable to be enumerated
    choice * cont;    // continuation for further distribution
public:
  naive(vector<var> vs,int i,choice * c ) : vars(vs), idx(i), cont(c) {}
  choice * choose(store * s, int i) {
    int l = s->getlo(vars[idx]); int h = s->gethi(vars[idx]);
    if (i==0) {
      s->tell(vars[idx],l,l);
      return (idx+1==vars.size() ? cont : new naive(vars,idx+1,cont));
    }
    else {
      s->tell(vars[idx],l+1,h);
      return new naive(vars,idx,cont);
    }
  }
}
```

```
  virtual choice * choose(store *, int)=0;
};
```

The member function `choose` of `choice` is given an integer i and returns its i^{th} alternative. Often, distributors fix one variable v of a given set of variables to a value x in the left child ($i = 0$) and exclude x from the domain of v in the right child ($i = 1$). Such distributors are called enumerators. Program 2 represents naive enumeration, where the variables of a given vector are enumerated from left to right, starting with the smallest values in their domains.

Note that `choose` returns new objects of class `naive`, until all variables in `vars` are fixed, in which case the continuation choice `cont` is returned. The continuation can be used to continue distribution with other constraints after all variables in `vars` are fixed, or to collect information from the encountered solutions. For example, if the continuation choice is an instance of the class `print` in Program 3, the lower bounds of the components of a given variable vector are displayed each time a choice is performed. The return value `NULL` indicates that the search is done.

4 Search

A technical difficulty in searching for solutions is that changes on a store done in one branch of the search tree may not affect the exploration of other branches. Two solutions to this problem are in use for this. The most common approach—following Prolog implementation tradition—is to mark a store and record all

Program 3 A Node Class for Printing Variables

```
class print : public choice {
private: vector<var> vars;
public:
  print(vector<var> vs) : vars(vs) {}
  node * choose(store * s, int i) {
    for (int j = 0; j < vars.size(); j++)
      cout<<"col: "<<vs[i]<<"\nrow: "<<s->getlo(vars[i])<<"\n";
    return NULL;
}
```

changes done after a mark. A backtrack operation allows to undo changes that were done since a given mark. The following operations on stores support this concept.

```
mark store::mark();
void store::backtrack(mark m);
```

A depth-first inference engine that employs backtracking is given in Program 4.

Using this engine, the first solution to the constraint problem of the previous section can be displayed by extending Program 1 as follows.

```
void main(int argc, char * argv[]) {
   store * s = new store();
   ...
   vector<var> vars(2);
   vars[1]=x; vars[2]=y;
   solve_one(s,new naive(vars,0,new print(vars)));
}
```

An alternative approach to backtracking is to copy stores at each node and start from the copy when a different branch needs to be explored. This technique

Program 4 Backtracking Depth-First Search

```
choice * solve_one(store * s,choice * c) {
  if (c == NULL) return c;
  mark m = s->mark();
  try {return solve_one(s,c->choose(s,0));}
  catch (Failure) {
    s->backtrack(m);
    return solve_one(s,c->choose(s,1));
  }
}
```

is described by Schulte in [Sch97b]. In [Sch99], Schulte analyses recomputation, a method for reducing the space consumption of copying-based search, and compares several variants of recomputation with backtracking-based search. In order to support copying/recomputation, we allow to copy stores through the following operation.

```
store * store::copy();
```

The implementation of depth-first search using the memory policy of copying unfortunately requires a re-implementation of depth-first search using `copy` instead of `mark` and `backtrack`. This hinders the reuse of algorithms. We would like to keep the aspects of exploration and memory policy independent from each other.

5 Memory Policy

The first step towards a modular search library is to separate the memory policy from the basic search algorithm. To this aim, we introduce a level of abstraction between engines and stores. This level is represented by abstract stores. An abstract store abstracts away how a store at a particular node in the search tree is represented.

```
class a_store {
public:
  virtual store * get_store()=0;
  virtual a_store * new_store(store);
}
```

A class representing an abstract store for backtracking is given in Program 5. Its instances keep a reference to a store. Initialization marks the store. Each time the store is retrieved via `get_store`, backtracking is performed and a new mark created. The member function `new_store` is a virtual constructor needed to create an abstract store of the same class as a given abstract store (see next section).

6 Search Trees

In order to describe tree search algorithms, it is useful to have a data structure for representing nodes of search trees. A node in a search tree is determined by an abstract store together with a distributor. The service provided by a node is to navigate to a child node. Assuming binary choices, a corresponding node class is given in Program 6. The creation of a child node needs to be protected from `Failure` exceptions arising from corresponding `choose` operations on choices. We represent leaf nodes of the search tree by the special nodes `success_node` and `failure_node`.

Program 5 Abstract Store with Backtracking Policy

```
class backtrack_store : public a_store {
  mark m;
  store * s;
public:
  backtrack_store(store * st) : s(st) {
    mark = st->mark(); }
  store * get_store() {
    s->backtrack(mark);
    mark = s->mark();
    return s;
  }
  // virtual constructor
  a_store * new_store(store * st) {
    return new backtrack_store(st);
  }
}
```

7 Exploration

The engine in Program 4 tightly integrates two aspects of a search engine, namely the exploration—the *order* in which nodes are visited—and the interaction—the *way* in which the exploration proceeds. The interaction is fixed to a straight first-solution search. Other possibilities are all-solution search, last-solution search (e.g. useful in combination with optimization); tools such as the Oz Explorer allow to interactively explore the search tree. Our aim is to separate these modes of interaction from the basic exploration algorithm. To this aim, we represent the basic exploration algorithm by a function `one_step` that says how to perform one step in the search. An interaction performs `one_step` as needed by the desired mode of interaction.

Program 7 shows a depth-first exploration class. A stack of nodes keeps track of the path from the root to the current node in the tree. Initially the stack contains the root node of the tree. If the stack becomes empty, `NULL` is returned. Otherwise, we pop the stack. If the top of stack is a non-leaf node, we check whether the left child was visited before. If not, we push the node back and create the left child. If it was visited before, we know that depth-first search explored the left subtree and thus create the right child. In both cases, the new node is pushed on the stack and returned.

An exploration can keep a representation of the tree in the state of the engine object. For depth-first search, we only need to keep the path from the current node to the root. This path is kept in a stack. Breadth-first search would keep instead of a stack a queue, representing the current fringe of the breadth-first tree.

Program 6 Nodes in Search Trees

```
class node {
private:
  choice * c;
  a_store * as;
public:
  node * left, * right;
  node(choice * ch, a_store * ast) :
    left(NULL), right(NULL), c(ch), as(ast) {};
  node * make_left_child() {
    store * s = as->get_store();
    try {
      choice * c0 = c->choose(s,0);
      left = c0 ? new node(c0,as->new_store(s)) : success_node;
    }
    catch (Failure) left = failure_node;
    return left;
  }
  node * make_right_child() {
    store * s = as->get_store();
    try {
      choice * c1 = c->choose(s,1);
      right = c1 ? new node(c1,as->new_store(s)) : success_node;
    }
    catch (Failure) right = failure_node;
    return right;
  }
}
node * success_node = new node(NULL,NULL);
node * failure_node = new node(NULL,NULL);
```

8 Interaction

Given an exploration, search can be conducted by repeatedly calling `one_step` and examining the resulting nodes. This task is performed by interaction functions. In the simplest case, an interaction searches for the first, the last or all solution nodes. For example, search for the first solution can be done as shown in Program 8.

Thus, putting it all together, the first solution to the problem given in Section 2 can be displayed by calling

```
void main(int argc, char * argv[]) {
  store * s = new store();
  ...
  first(new depth_first(new backtrack_store(s),
                        new naive(vars,0,new print(vars))));
}
```

Program 7 Depth-first Exploration

```
class depth_first : public exploration {
private:
  Stack<node *> stack;
public:
  depth_first(a_store * s,choice * c) {
    stack.push(new node(c,s));
  }
  node * one_step() {
    if (stack.empty()) return NULL;
    node * n = stack.pop();
    if (n == success_node || n == failure_node) return n;
    node * n2;
    if (!n->left) {        // there's a no left child yet
      stack.push(n);       // go left
      n2 = s->make_left_child();
    }
    else
      n2 = s->make_right_child();  // go right
    stack.push(n2);
    return n2;
  }
};
```

Program 8 First-solution Search

```
node * first(exploration * e) {
  node * n;
  while (n = e->one_step()) if (n==success_node) break;
  return n;
}
```

9 STK: A Search Toolkit for Oz

The presented architecture allows to develop classes for memory policy, classes for exploration and interaction functions independently from each other. We used the presented concepts to develop a modular search toolkit, STK, for the constraint programming language Oz. STK is available in [CH99] and documented in [Che99]. In addition to the dimensions of memory policy, exploration and interaction, STK supports the design dimensions optimization, information and visualization, which are briefly described below.

9.1 Optimization

Modules of dimension "optimization" allow to modify an engine such that a pruning behavior is achieved. A well-known method for pruning is branch-and-bound.

After a solution is found, a constraint is added to every subsequent node that the next solution should be better than the one already found, where "better" is defined using a given optimization function. In addition to branch-and-bound, STK provides a module for restart optimization, where the constraint is added to the root node and search restarts from the root.

Optimization modules rely on interaction modules to indicate that a solution has been found.

9.2 Information

When designing constraint-based solutions, it is often useful to inspect the search tree in detail. The dimension "information" provides modules for accessing the information in a given node in the tree. Using an information module, interaction modules and a visualization modules can be enhanced by the facility of application-specific display of the information in a node, which is passed to the information module in form of an Oz procedure.

In addition, STK provides an information module `edgeInformation` to display information on the distribution step that leads to a given node. For example, it is possible to display which variable was enumerated and to which value it was bound. To achieve this behavior, distributors communicate with the STK engine to annotate choice points with information that is then stored in nodes and displayed on demand. STK provides a suitably modified version of Mozart's distributor library `FD.distribute`.

9.3 Visualization

The visualization of search trees is a dimension orthogonal to exploration and interaction. In a corresponding module, the creation of the root node leads to opening of a display area in which the root node is graphically represented. Similar to the Oz Explorer, this module uses an incremental tree drawing algorithm inspired by [Ken96]. Figure 1 shows the display of an engine that combines breadth-first search with visualization.

STK provides two visualization modules. The first, `standardDisplay`, requires the `tracer` interaction and allows to interactively explore the search tree, whereas the second, `simpleDisplay` only displays the search tree as it is being explored, but works with any interaction module.

9.4 Generating Inference Engines

Figure 2 shows the currently available modules for each dimension available in STK.

In STK, each module is described by two classes; a node class that specifies the local effect of the module and an engine class that specifies its global effect. In order to achieve generation of custom-made search engines at runtime, we need to build two inheritance graphs containing the desired node and engine

Fig. 1 Visualization of Breadth-First Search. Choice nodes are represented by circles, solutions by diamonds and failure nodes by squares.

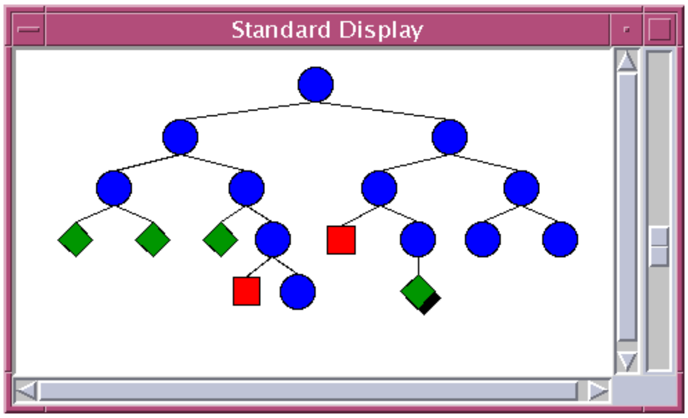
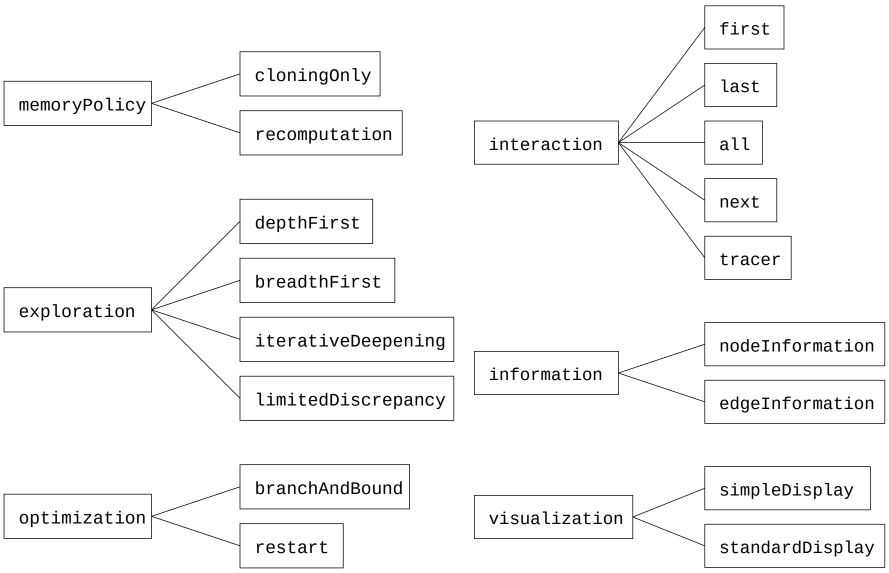


Fig. 2 Dimensions and Modules of STK



classes for each dimension. This results in different inheritance graphs for every desired combination of search algorithm with extension modules. Here, we are making use of Oz's advanced object-oriented features [Hen98]. Classes are first-class citizen in Oz, which means that we can decide at runtime from which parent class a class should inherit. This feature allows us to define a function `MakeEngine` to which we pass the desired modules as arguments.

```
E={STK.makeEngine dims(exploration:STK.depthFirst
                       interaction:STK.last
                       optimization:STK.branchAndBound)}
```

Here the modules `STK.depthFirst` and `STK.branchAndBound` contain engine and node classes, describing their global and local behavior. The function `MakeEngine` constructs from these classes the final classes from which the desired inference engine is constructed. A graphical front-end to the function `MakeEngine` as shown in Figure 3 allows the selection of the desired modules and creates a corresponding engine upon button click.

9.5 Example

Through engine generators, STK supports experiments with different combinations of search algorithms and extensions. Figure 4 shows a snapshot of an engine constructed by the engine generator in Figure 3, which combines limited discrepancy search [HG95] with recomputation, node-level tracing, edge information, visualization, recomputation and branch-and-bound optimization. The tracer tool on the left displays the edge information of the three edges leading to the left-most solution of a crypto-arithmetic problem. Using STK, engines can be tailor-made for individual applications and embedded in their graphical user interface.

New modules such as application-specific explorations and search tree visualizers can be easily added to STK.

10 Related Work

Individual aspects of this work are addressed by other systems. The most recent version of Ilog Solver [ILO99a] provides object-oriented abstractions for pro-

Fig. 3 An Interface for an Engine Generator

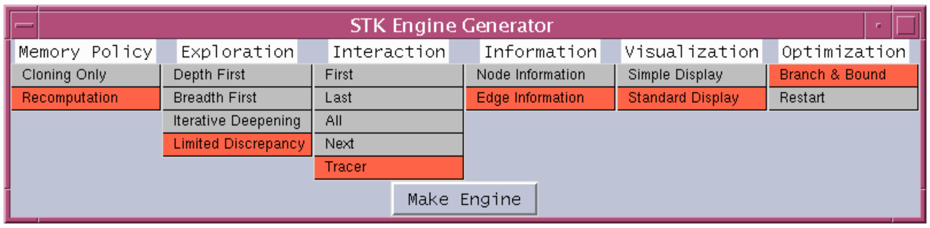
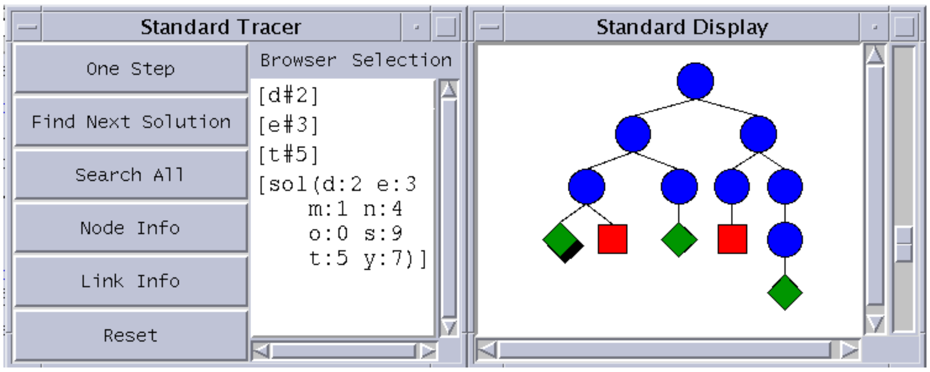


Fig. 4 A Custom-made Engine



gramming search engines. Tracing facilities for constraint-based search are investigated by [Mei95] and provided by the tool OPL Studio [ILO99b]. Visualization of search trees together with tracing is provided by the Oz Explorer [Sch97a], where earlier tools are also discussed. The Mozart system provides several different search engines, including limited discrepancy search and parallel search. None of these systems address the question how to systematically reuse components of search engines.

Compared to the engines provides by the Mozart system, our modular approach carries a certain overhead, due to late binding and extra member function calls. However, since for typical applications, the majority of the overall runtime is spent on propagation and cloning/recomputation, this overhead is usually negligible. Benchmarks given in [Che99] show that STK is competitive in performance with the Oz search libraries except for the search tree visualization, where the Oz Explorer is currently significantly faster.

The facility of annotating choice points and display the annotation on demand provided by the module `edgeInformation` in Section 9.2 is not available in the Oz Explorer, but could be supported with few changes.

11 Conclusion

We identified the following dimensions for designing constraint-based inference engines: memory policy, exploration (defining the search algorithm), interaction, information, visualization and optimization. This structure provided the base for an object-oriented software design that supports flexible reuse and recombination of components. Evidently, we do not claim that this list of dimensions is exhaustive. We argued, however, that it is useful to identify such dimensions in order to obtain a modular design of search libraries.

We outlined how a constraint programming library or language can utilize the secribed design. The toolkit STK [CH99] for search engines in Oz provides evidence for the practicality of the approach.

Besides improving the performance of STK for visualization of search trees, future work includes the development of a C++ constraint programming library based on the presented design.

Acknowledgements

Gert Smolka collaborated on the development of the room concept and corresponding abstractions in an ML setting, which provided a blueprint for stores in C++. Tobias Müller supported us by explaining the implementation of the Mozart constraint programming support. The project benefited from a travel grant from the National University of Singapore (project ReAlloc) and the hosting of the third author by the Programming Systems Lab, Saarbrücken.

References

[CH99] Tee Yong Chew and Martin Henz. SearchToolKit: A toolkit for constraint-based tree search. Oz code available via WWW at `http://www.comp.nus.edu.sg/~henz/projects/stk`, 1999.

[Che99] Tee Yong Chew. A toolkit for constraint-based tree search. Honours Year Project Report, School of Computing, National University of Singapore, available at `http://www.comp.nus.edu.sg/ henz/projects/toolkit/`, March 1999.

[CL96] Yves Caseau and François Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming*. TU Berlin, 1996.

[Hen98] Martin Henz. *Objects for Concurrent Constraint Programming*. The Kluwer International Series in Engineering and Computer Science, Volume 426. Kluwer Academic Publishers, Boston, 1998.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers, San Mateo, CA.

[ILO99a] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.4, Reference Manual*, 1999.

[ILO99b] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *OPL Studio User Manual*, 1999.

[Ken96] Andrew J. Kennedy. Functional pearls: Drawing trees. *Journal of Functional Programming*, 6(3):527–534, May 1996.

[LP99] Irvin J. Lustig and Jean-François Puget. Program != program: Constraint programming and its relationship to mathematical programming. white paper of Ilog Inc., Mountain View, CA 94043, USA, available at `http://www.ilog.com`, 1999.

[Mei95] Micha Meier. Debugging constraint programs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 976, pages 328–344, Cassis, France, September 1995. Springer-Verlag, Berlin.

[Moz99] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[Pug94] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, November 1994.

[Sch97a] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press, Cambridge, MA.

[Sch97b] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of the International Conference on Logic Programming*, 1999. to appear.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.