# A Toolkit for Constraint-based Inference Engines

Martin Henz and Chew Tee Yong
School of Computing
National University of Singapore
Singapore 117599
email: {henz,chewteey}@comp.nus.edu.sg

April 16, 1999

**Abstract**

Solutions to combinatorial search problems can benefit from custom-made constraint-based inference engines that go beyond depth-first search. The concurrent constraint language Oz provides support for programming inference engines. The Mozart system for Oz comes with several engines, extended in dimensions such as interaction, visualization, and optimization. However, these 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. The SearchToolKit, a public domain 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. Almost all programming systems for CP(FD) are based on Prolog, including libraries such as Ilog Solver [Pug94], in a sense that Prolog's depth-first search is built-in as their only inference engine. In these systems, the programmer is limited to using depth-first search.

Exceptions are the languages CLAIRE and Oz. CLAIRE [CL96] supports the programming of backtracking-based inference engines by operations on a global constraint database. Oz [Smo95] allows to encapsulate search in 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 the search tree.

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 of the Mozart inference engines, 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, this 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 will present a design that allows to plug together search algorithms with modules for optimization, interaction, visualization, etc. If a search algorithm is constructed following certain guidelines, it becomes trivial to equip it with branch-and-bound optimization, node-level tracing, Oz-Explorer-like visualization of the search tree, and other features. Modules for individual dimensions can be developed independently and employed by all search algorithms.

## 2 Scripts, Spaces, Engines

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

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

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 the ones given in the problem. Constraints are operationalized using propagators, which are connected to the store. Propagators observe the domains of their variables and amplify 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`.

The principle of finite domain constraint programming can be summarized as "propagate and distribute". After exhaustive propagation, a distribution step is done, i.e. a constraint $c$ is chosen and search continues 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 $\neg c$ make up a choice point.

The problem together with suitable distributors form a script, a procedure that describes the search tree of the problem. For example, the script for the problem above may look in Oz as follows.

```
proc {Script Vars}
   [X Y]=Vars
in X :: 5#10     Y :: 1#7
   X <: Y        X + Y >=: 11
   {FD.distribute naive Vars}
end
```

where the library distributor `FD.distribute` specifies that the variables `V` in `Vars` are used from left to right in order to generate constraints $c$ of the form `V =: I`, where `I` is the lowest value in the current domain of `V`.

While a script describes a search tree, it is the task of engines to explore that tree. Engines are programmed in Oz using the abstraction of spaces. Spaces are concurrent entities that execute a script. In order to perform a distribution step, a distributor in the script may install a choice point, after which the script suspends its execution. Oz provides several operations on spaces:

- The procedure `NewSpace` gets a script as argument and executes it.
- The procedure `Ask` waits until the script in the space stops its execution and returns whether the space is failed, has encountered a choice point, or is otherwise succeeded.
- The procedure `Commit` allows to commit the current choice point of the space to one of its alternatives (for the sake of simplicity, we assume here that there are always two alternatives).

- The procedure `Clone` allows to create an identical copy of a space.

Further primitives allow to access the values of the root variable of the script and to add further constraints. The operations on spaces are described in detail in [Sch97b]. Schulte [Sch99] shows that constraint-based search using copying, combined with a recomputation policy, can be more efficient than the standard backtracking-based approach.

The following example shows how a simple copying-based depth-first search engine can be programmed using `Ask`, `Commit` and `Clone`.

```
fun {DepthFirst S}
  case {Ask S}
  of failed then nil              % no solution
  elseof succeeded then S         % solution found
  elseof alternatives then
    C={Clone S} in
    {Commit S 1}
    case {DepthFirst S} of nil % no solution to the left
    then {Commit C 2}             % use clone to go right
         {DepthFirst C}
    elseof T then T               % solution to the left
    end
  end
end
```

This engine can be used to compute a space that contains the first solution to our example problem by the program

```
SolutionSpace={DepthFirst {NewSpace Script}}
```

In practice, the implementation of search engines is complicated by various requirements in addition to the basic search algorithm. The search tree often needs to be pruned using a branch-and-bound strategy, solutions need to be generated one-by-one, the search tree needs to be visualized and so on.

# 3 Overall Design

Our goal is to separate the basic search algorithm from such enrichments, whose implementations we call modules, in order to obtain maximal flexibility in programming and using search engines.

If search algorithms are to be enriched by new behavior through modules for interaction, visualization, optimization and the like, the algorithms need to systematically provide entry points to the corresponding modules. The concept of late binding in object-oriented programming is designed for this purpose. Oz provides a powerful object system [HSW95, Hen98] such that object-oriented programming can be conveniently put to work for this task.

A crucial observation for the overall design of the library is the distinction between local and global effects. Operations that affect the whole search tree such as finding the next solution are global whereas operations that effect only a single node in the search tree and its immediate neighborhood are local. It turned out to be useful to make this distinction explicit in the design of the library. Thus every module comes with two classes: an engine class and a node class. Modules are ordered according to their dependency from one another. A module $m_1$ depends on another module $m_2$, if one of $m_1$'s classes needs to override a method of a corresponding class of $m_2$.

In order to generate an engine, we build an engine class and a node class by inheriting from the desired modules that implement respective dimensions in the order of their dependency. For example, let us assume that we want to generate an engine with module `depthFirst` for dimension "exploration", module `last` for dimension "interaction", and module `branchAndBound` for dimension "optimization". If the dimension "optimization" depends on "interaction", Figure 1 depicts a possible inheritance relationship for the desired engine and node class. Note that the classes `BaseNode` and `BaseEngine` provide basic functionality common to every engine, and that the classes `FinalNode` and `FinalEngine` represent the classes from which the desired engine can be constructed.

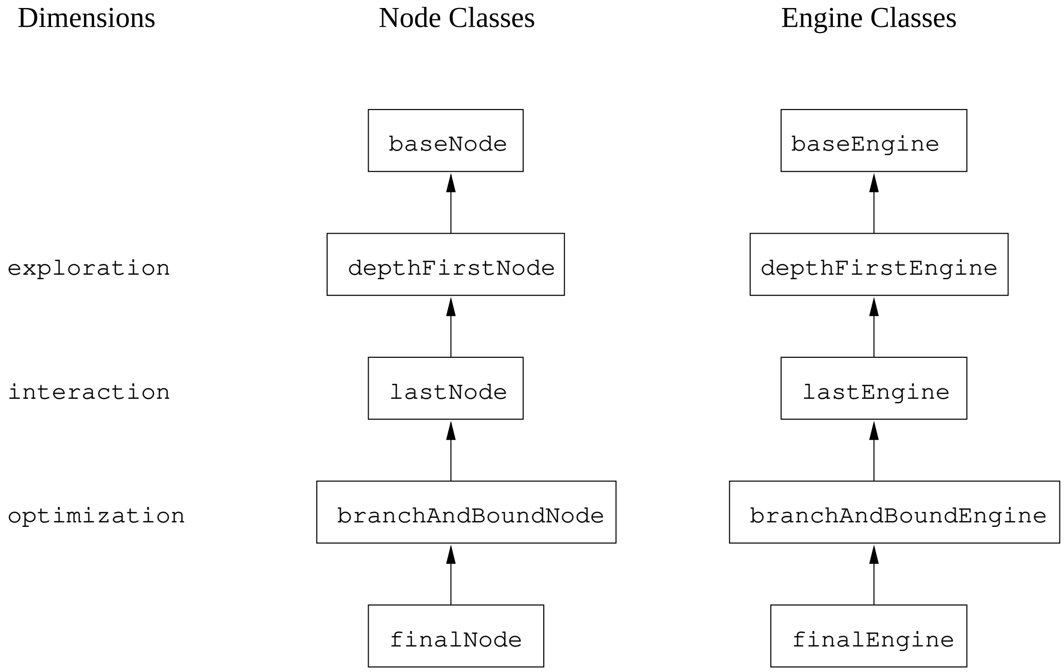
# 4 Dimensions

In this section, we describe each of the identified dimensions in detail. For each dimension, we briefly outline the corresponding modules provided by the public domain library SearchToolKit (STK) [Che99, HC99].

## 4.1 Memory Policy

A node of a search tree is represented by a node object. In its simplest form, a node object encapsulates a space and provides the operations `NewSpace` and `Ask` on spaces through object invocation. For example, instead of calling the procedure `Ask` as in `Answer={Ask MySpace}`, the method `ask` can be invoked on a node object as in `Answer={MyNodeObject ask($)}`. Nodes are tailor-made for tree search and provide a method `createChild(N $)` for creating a child node of a given node. In its simplest form, this operation combines the operation of cloning and committing in a single method. The method

**Figure 1** Example for Inheritance of Modules. The arrows express the "inherits from" relationship between classes.




invocation `MyChild={MyNodeObject createChild(1 $)}` returns the left child node which contains a clone of the space of `MyNodeObject` committed to the first alternative of its choice point.

Cloning spaces for every non-deterministic step can be excessively memory-consuming. The dimension "memory policy" allows to modify this behavior such that parent nodes may pass their space to a child node in the method `createChild(N $)`. When the parent needs its own space again, it recomputes it from the closest ancestor that owns a space by cloning that space and reinstalling the appropriate distribution constraints. This technique is called recomputation and described in detail by Schulte in [Sch97b].

By encapsulating the recomputation behavior of a node in a module implementing the dimension "memory policy", the memory policy can be hidden away from other modules. These modules simply call the method `createChild(N $)` without being aware of the operations on spaces that this triggers. STK provides the modules `cloningOnly` and `recomputation` as implementations of the dimension "memory policy".

**Program 1** An Exploration Module for Depth-first Search

```
class DepthFirst
  attr stack:nil                  % stack initially empty
  ⋮
  meth oneStep($)
    case @stack of nil then nil  % stack empty; no solution
    elseof Node|RestStack in
      case {Node ask($)}
      of alternatives
      then if {Node getChild(1 $)} == unit
           then Child={Node createChild(1 $)} in
                stack <- Child | @stack
                Child
           else Child={Node createChild(2 $)} in
                stack <- Child | RestStack
                Child
           end
      else stack <- RestStack     % failure or success
           {self oneStep($)}
      end
    end
  end
end
```

## 4.2 Exploration

The dimension "exploration" in a search engine describes the the search algorithm to be employed. A module of dimension "exploration" is implemented through an engine method `oneStep`, which returns the next node to be examined. Repeated execution of `oneStep` leads to the desired exploration of the search tree. For example an exploration module `depthFirst` can be defined as shown in Program 1.

Initially the stack contains the root node of the tree. If the stack becomes empty, the next node is not defined (`nil`). If the top of stack is a non-leaf node, we distinguish whether the left child was visited before. If not, we create the left child, push it on the stack and return it. If the left child was visited before, we create the right child, replace the top of stack by that child and return it. If the top of stack is a leaf node (failure or success), it is popped and `oneStep` is called recursively.

An exploration can keep a representation of the tree in the state of the engine object. Thus, `oneStep` can be much more complex than in this example,

**Program 2** An Interaction Module for First Solution Search

```
class FirstSolution
   attr script:nil
   ⋮
   meth first($)
      case {self oneStep($)}
      of nil then nil                       % no solution
      elseof N then
         case {N ask($)}
         of succeeded
         then {self solutionFound(N)} N % solution found
         else {self first($)}            % search continues
         end
      end
   end
end
```

where we only need to know the path from the current node to the root. This path is kept in a stack. For example, breadth-first search would keep instead of a stack a queue, representing the current fringe of the breadth-first tree.

Apart from depth-first and breadth-first search, STK provides exploration modules for limited discrepancy search [HG95] and iterative deepening [Kor85]. Explorations require node methods such as `ask($)` and `createChild(N $)` and provide the method `oneStep($)`. Such interface information defines the dependency structure of the modules and is described in detail in [Che99].

## 4.3 Interaction

Given a search algorithm, search can be conducted by repeatedly calling `oneStep` and examining the resulting nodes. Modules of dimension “interaction” perform this task. 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 2.

Interactions can provide for incremental search at different granularities. STK provides interactions for solution-by-solution search through the interaction module `next`. A finer granularity is provided by the interaction module `tracer` which allows to perform search step-by-step, similar to the Oz Explorer but without tree visualization.

## 4.4 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 a 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 the distributor library `FD.distribute`.

## 4.5 Visualization

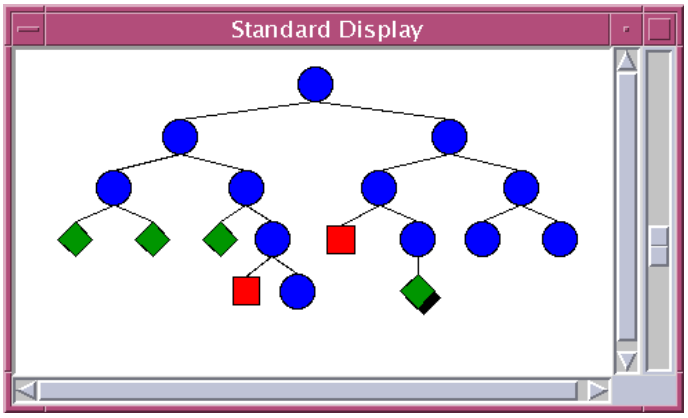
The visualization of search trees is a dimension orthogonal to exploration and interaction. In a corresponding module, the creation of the root node leads to opening of a display area in which the root node is graphically represented. The node class of a visualization module refines the method `createChild(N $)` such that a new graphical object corresponding to the new node is drawn. Similar to the Oz Explorer, this module uses an incremental tree drawing algorithm inspired by [Ken96]. Figure 2 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.

## 4.6 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.

**Figure 2** Visualization of Breadth-First Search. Choices are represented by circles, solutions by diamonds and failure nodes by squares.




Optimization modules rely on interaction modules to indicate that a solution has been found, as done by the interaction `first` in Section 4.2 via the engine method `solutionFound(N)`.

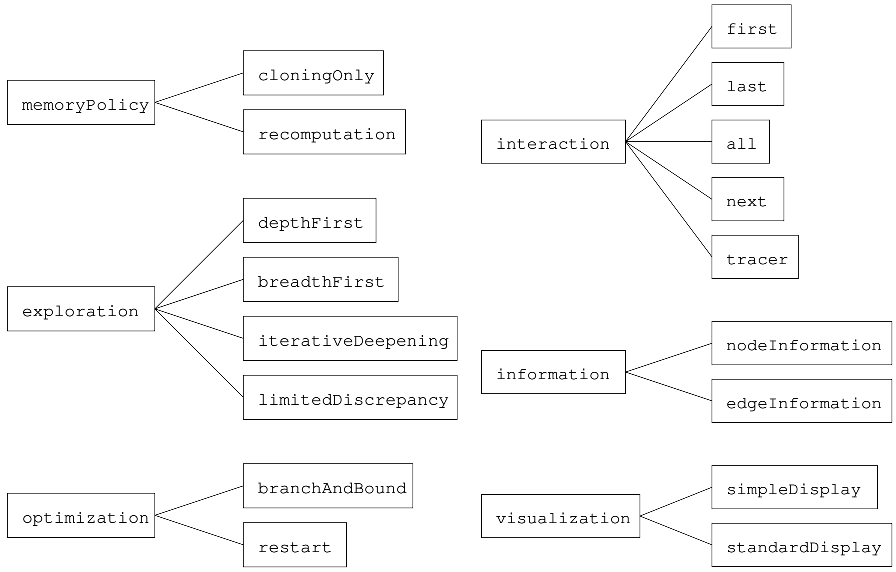
# 5 Generating Inference Engines

Figure 3 shows the currently available modules for each dimension. In order to construct an engine, we need to build two inheritance graphs containing the desired node and engine classes for each dimension. This results in different inheritance graphs for every desired combination of search algorithm with extension modules. In usual object-oriented languages such as C++ and Java, it would be necessary to recompile the library in order to get a specific combination of modules. Instead, we are making use of Oz's advanced object-oriented features. 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.

**Figure 3** Dimensions and Modules of STK




In order to achieve this behavior, modules such as `STK.branchAndBound` are functions that take a record `Super` as argument, which contains a node class `nodeClass` and and engine class `engineClass`. The function returns a similar record, constructed by inheriting from `Super`'s components, as follows.

```
fun {STK.branchAndBound Super}
 desc(nodeClass:   class $ from Super.nodeClass   ··· end
      engineClass:class $ from Super.engineClass··· end
end
```

Now `MakeEngine` only needs to compose these functions in an order that is consistent with the dependencies of the corresponding modules. It is easy to put together a graphical front-end to the function `MakeEngine` as shown in Figure 4 that allows to select the desired modules and create a corresponding engine upon button click.

An alternative approach to the described design would have been to modularize using objects, not classes. In such a design, an engine object has access to one component object per dimension. Such a design does not require first-class classes, but makes it more difficult to add dimensions since the dimension to which operations belong have to be made explicit. The design of STK seems to fit the concepts of aspect-oriented programming [KLM+97] and it would be interesting to formulate it using this paradigm.

**Figure 4** An Interface for an Engine Generator

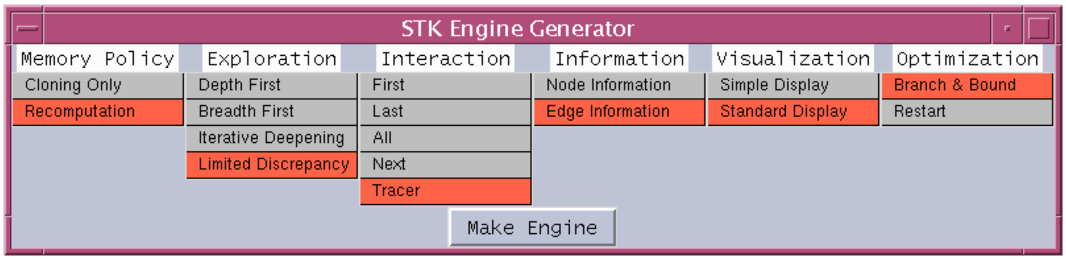
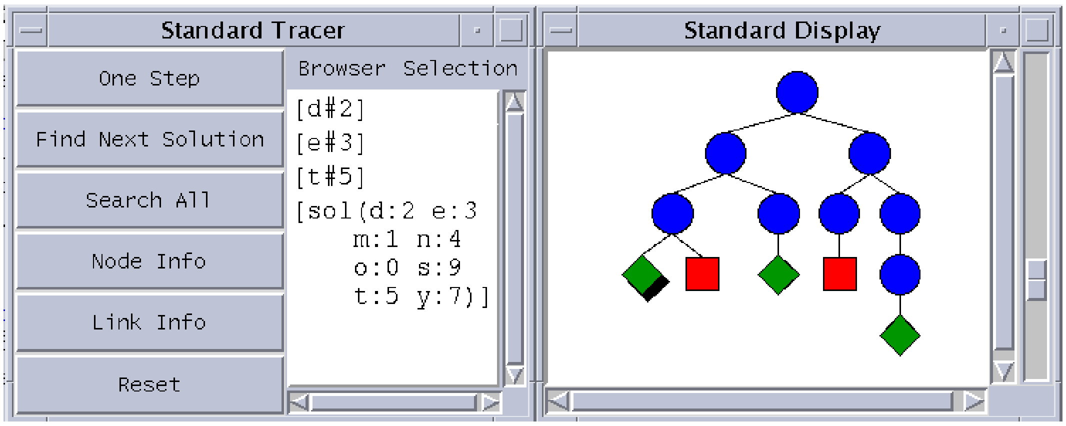



**Figure 5** A Custom-made Engine




# 6 Example

Through engine generators, STK supports experimentation with different combinations of search algorithms with extensions. Figure 5 shows a snapshot of an engine constructed by the engine generator in Figure 4, 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.

# 7 Related Work

Individual aspects of this work are addressed by other systems. In recent versions, the Ilog Solver [ILO97] provides a class `IlcManager` that allows to program search facilities in an object-oriented manner. However, the Ilog Solver is limited to variations of depth-first search and does not support node-level tracing or visualization of the search tree. Tracing facilities for constraint-based search are investigated by [Mei95]. Visualization of search trees together with tracing is provided by the Oz Explorer [Sch97a]. 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, STK carries a certain overhead, due to late binding and extra method calls. However, since for typical applications, the majority of the overall runtime is spent on propagation and cloning/recomputation, this overhead is typically 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 4.4 is not available in the Oz Explorer, but could be supported with few changes.

# 8 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.

The separation between engine classes and node classes allowed to distinguish between local and global issues in search.

The design relies on late binding and benefits from first-class classes, both supported by the constraint language Oz. The presented architecture provides the base for the SearchToolKit, a public domain search library for Oz. We demonstrated the flexibility of STK by showing how a non-standard search algorithm, limited discrepancy search, can be equipped by simple steps with branch-and-bound optimization, visualization and recomputation. The performance of STK is close to Mozart's search libraries, except for visualization.

# References

[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.

[HC99] Martin Henz and Tee Yong Chew. SearchToolKit: A toolkit for constraint-based tree search. Oz code available via WWW at `http://www.comp.nus.edu.sg/~henz/projects/stk`, 1999.

[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.

[HSW95] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[ILO97] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, Reference Manual*, 1997.

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

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In J. Bosch and S. Mitchell, editors, *Proceeedings of the European Conference on Object-Oriented*

*Programming*, Lecture Notes in Computer Science 1357, Jyväskylä, Finland, 1997. Springer-Verlag, Berlin.

[Kor85] Richard E. Korf. Depth-first iterative deepening. *Artificial Intelligence*, 27(1):97–109, 1985.

[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.