# Solver — An Oz Search Debugger

Christian Schulte

*Abstract*— **This paper shows the Oz Solver, a tool designed to support the development of constraint programs.**

**The Solver allows the user to interactively solve a constraint problem. As the problem's search tree is explored the Solver incrementally draws its graphical presentation. The user has access to the computation state at any node in the tree.**

**Although search is controlled interactively by the user, implementing the Solver does not require any other primitives than those provided by Oz.**

**It is planned to include the Oz Solver in the next release of the DFKI Oz system.**

*Keywords*— **Constraint Programming, Programming Environment, Debugging, Encapsulated Search, Graphical User Interfaces.**

## 1. Introduction

Oz [13], [4], [10], [14] is a concurrent constraint language developed at DFKI supporting higher-order functional programming, concurrent object-oriented programming, and constraint programming.

This paper presents the Oz Solver, a tool designed to support the development of constraint programs. The development of constraint programs is substantially different from the development of, say, functional or concurrent object-oriented programs. This has to be taken into account by the tool.

The Solver chooses the metaphor of a search tree to follow the execution of constraint programs. Nodes in the search tree represent choices, failures, and solutions. The Solver allows to explore parts of the search tree interactively. The search tree is drawn incrementally as it is explored. Non-failed nodes in the explored parts of the tree carry information of the corresponding computation state. In addition to predefined functionality to display the information upon access, users can define their own display procedures.

Using a problem solver in Oz is factorised into two orthogonal components: specifying the problem and the search strategy to solve the problem. Rather than providing a single built-in search strategy, in Oz different search strategies can be obtained from a single primitive, called solve combinator [10], [11], [3], [7]. The Oz Solver uses this primitive to implement an interactive search strategy. Instead of using a search strategy like depth-first or branch-and-bound search, the Oz Solver can be used without making any changes to the problem to be solved. The Solver is written completely in Oz, which testifies the expressiveness of its search primitive.

Meier describes in [6] the Grace tool which is built on top of the Prolog System ECLiPSe. The Grace tool is intended to support the development and debugging of finite domain constraint programs. However, it does not use the metaphor of a search tree. Instead it maintains and displays a backtracking history of the finite domain variables involved. There is no possibility to interactively explore the search space. In contrast to the Solver it allows also to trace the constraint propagation process. The display of information is very detailed, but fixed. The user has no possibility to parametrise the tool with own graphical display tools. To make use of the Grace tool the user has to modify its programs.

The plan of the paper is as follows. Section 2 introduces and explains some basic notions of constraint programming in Oz. The development process of constraint programs is sketched in Section 3. The next section shows an example of a constraint program and introduces how the Solver is used. Section 5 shows the features of the Solver and explains how they can be used. Section 6 gives the structure of the implementation of the Solver. The core of the implementation, the creation of the search tree, is presented in Section 7. Section 8 gives a quick overview how a layout for an search tree can be computed.

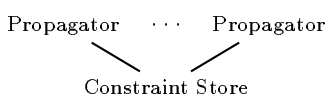
## 2. Constraints, Propagators, and Search

Constraint programs make use of constraints, propagators, and search. To give a quick overview on constraint programming, we show how to solve finite domain constraint problems. However, the presented ideas carry over to solving other constraint problems.

Only primitive constraints can be written to the constraint store. A primitive finite domain constraint in Oz has the form $x \in D$ where $D$ is a finite subset of the positive integers. To these constraints we refer to as *basic* constraints. Other basic constraints which make sense in the context of integer valued variables are the constraints $x = y$ and $x = n$, where $n$ is a positive integer.

More expressive constraints, e.g., $x + y = z$ or $x * 2 = y$, are not written to the constraint store. This restriction guarantees that entailment and satisfiability of constraints can be checked efficiently. These constraints are called *nonbasic*.

Nonbasic constraints are accommodated in Oz as concurrent tasks called *propagators*. This yields the following picture:




A propagator tries to amplify the store. A propagator $P$ can amplify a constraint store $S$ by writing a basic constraint $\phi$ to $S$, if $S \wedge P$ entails $\phi$, but $S$ does not entail $\phi$. The propagator remains in the computation space until it

Christian Schulte is with the Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany.
E-mail: schulte@dfki.uni-sb.de

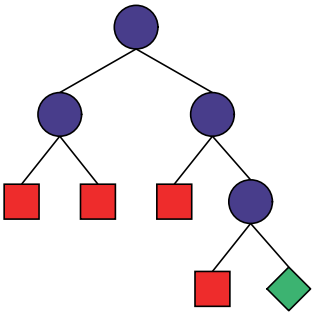


Fig. 1. Search tree created by interleaving constraint propagation and choice point creation.

is entailed by the constraint store. In this case, all information represented by the propagator has been made explicit by writing it to the constraint store. The process of making information explicit by writing it to the constraint store is called *constraint propagation*.

However, a constraint problem cannot be solved by constraint propagation only. If no further amplification of the constraint store is possible, the computation space becomes *stable*. If a computation space becomes stable, it might be the case that there are variables with more than one possible integer value. Now search comes into play. In this situation, an appropriate variable $x$ is chosen to be enumerated, which means to try each of its values still possible separately. This is achieved by creating a *choice point* for the variable $x$. The alternatives of the choice point constrain $x$ to its different possible values.

Following a particular alternative means to assign the variable a value. Writing the assignment, i.e., a basic constraint $x = n$, to the constraint store, amplifies it, which in turn can enable some propagators to even further amplify the store.

This search process of interleaved constraint propagation and choice point creation is repeated until either a solution is obtained or failure occurs. A solution means that all propagators have disappeared, and have written their information completely to the store. In case failure occurs, the next alternative of the previously created choice point is tried.

Solving a constraint problem means to initially set up some basic constraints and propagators and then follow the interleaved process of constraint propagation and choice point creation until a solution is found. This search process can be displayed as shown in Figure 1 as a search tree. Choice points in the tree are shown as circles, failed nodes as rectangular boxes and solution nodes as diamonds. One can think of the lines connecting the nodes as showing where constraint propagation takes place.

## 3. Developing Constraint Programs

The development of constraint programs is quite different from the development of functional or concurrent object-oriented programs. Although being different in nature, the previous section makes clear that the execution of constraint programs follow a certain, well-defined scheme. Interleaving constraint propagation and search gives a nice model of how the execution of constraint programs can be traced and visualised. The natural metaphor for this seems to be that of a search tree as exemplified by Figure 1.

The design of a constraint program can be divided into three major steps:

1. Design of the data structures hosting the variables to be constrained.
2. Describing properties of the solution by stating constraints on the variables.
3. Fixing an enumeration strategy, which chooses the right variable for choice point creation.

Existing tools support some aspects of these tasks. For instance, the Browser [8] can monitor the creation of the data structures and variables. But even though the data structures might have been created correctly, the problem might not have a solution, or finding a solution might take an intolerably long time.

This makes development and debugging of constraint programs very different compared to the debugging of functional and concurrent object-oriented programs. In the latter two cases it is possible to decide at any step of the execution whether the execution state is correct or not. In contrast to this, in order to decide if a particular execution state of a constraint program is correct, we have to know whether this state eventually leads to a solution of the problem. But the situation is even worse. Constraint programs normally do not have a clear performance model, in contrast to non-constraint programs. Even if a problem has a solution, the program might be impractical, if its performance is drastically poor.

Tools to support the development of constraint programs have to take these issues into account. Tasks that the tool must support include the following:

- Finding a right solution, that is the constraints stated initially might have been simply wrong.
- Finding redundant constraints such that more constraint propagation can take place, which reduces the size of the search tree.
- Finding an appropriate enumeration strategy which takes some problem specific properties into account.

The Solver supports these tasks by making it possible to interactively explore the search tree and to visualise parts of it as soon as they are explored. Each node in the explored part of the search tree carries information on the computation state. That is, for solved nodes the solution is provided as information, whereas for choice nodes, the constraints and propagators from the computation space just before choice point creation are provided.

## 4. Example: Aligning for a Photo

Five people want to make a group photo. Each person can give two preferences next to whom he or she wants to be placed on the photo. Our task is to write an Oz program that computes a placement of the people such that as many preferences as possible are satisfied.

```
Names = [alice bert chris deb evan]
Prefs = [alice#chris bert#evan chris#deb
         chris#evan deb#alice deb#evan
         evan#alice evan#bert]

proc {Photo Sol}
   Pos = {FD.record pos Names
                    1#{Length Names}}
       = {FD.distinct}
   Ful = {Map Prefs
          fun {$ A#B}
             (Pos.A+1 =: Pos.B) +
             (Pos.A-1 =: Pos.B) =: 1
          end}
   Sat = {FD.int 0#{Length Prefs}}
       = {FD.sum Ful ´=:´}
in
   Sol = Pos#Ful#Sat
   {FD.enum.naive Pos}
end
```

Fig. 2. An Oz program to solve the photo alignment problem.

An Oz program that solves the photo alignment problem is shown in Figure 2[1]. The problem is stated as the unary procedure `Photo`, where its argument `Sol` is constrained to the solution of the problem. The names of the persons and their preferences are given to us by the respective lists. The record `Pos` maps the person's name to a position, which is a finite domain between 1 and 5. All fields of `Pos` are enforced to be distinct by the propagator `FD.distinct`. The list of preferences is mapped to a list `Ful` of finite domain variables between `0` and `1`, such that each of its elements is either `1` in case the preference can be fulfilled or `0` otherwise. The overall satisfaction `Sat` is given by adding all elements of `Ful`.

The mapping of a preference to a finite domain variable makes use of so-called reified propagators. A reified propagator employs a control variable. If the propagator is entailed (disentailed), then the control variable is constrained to `1` (`0`). If the control variable is `1` (`0`), then the constraint of the reified propagator is enforced (the negation is enforced). The reified propagator `Pos.A+1=:Pos.B` expresses that `A` is placed to the right of `B`, whereas `Pos.A-1=:Pos.B` expresses that `A` is placed to the left of `B`. Thus, the control variable of the reified propagator stating that the sum of both is `1`, yields `1` if `A` and `B` are placed next to each other, and `0` otherwise.

Now we have to apply search to solve the `Photo` problem. For this we give the Solver the problem and an order describing that we are looking for the solution with maximal satisfaction:

```
{Solver query(Photo
              proc {$ 0 N} 0.3 <: N.3 end)}
```

The Solver pops up a window, showing a single choice node (drawn as a circle). Asking for the next solution, and the Solver draws a search tree and stops at the first solution. Now the Solver looks as shown in Figure 3, where a solved node is diamond-shaped.

[1]The program has been adopted from an Oz program written by Gert Smolka who has been inspired from a CHARME program in [1].

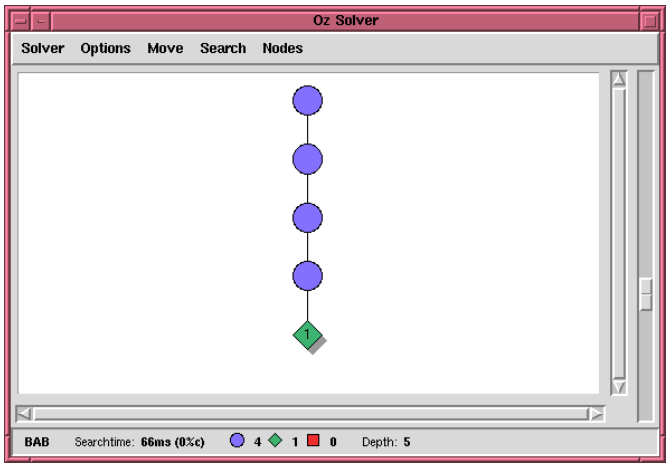



Fig. 3. The Solver after invoking search for a next solution.

Double-clicking the solution node shows the solution in the Browser. The first solution looks as follows in the Browser :

```
pos(alice:1 bert:2 chris:3 deb:4 evan:5) #
[0 0 1 0 0 1 0 0] # 2
```

The output is fairly difficult to understand. Suppose we have a procedure `DrawPhoto`, which presents information attached to the nodes in an appealing graphical form. By feeding the expression

```
{Solver add(information:DrawPhoto)}
```

the Solver is configured such that double-clicking a node applies the `DrawPhoto` procedure to the node's information. Figure 4 shows from top to bottom the information

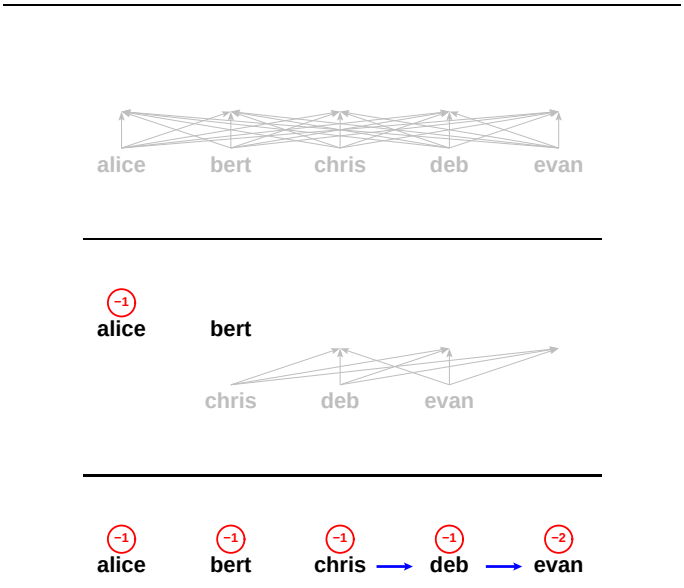



Fig. 4. Graphical output for the photo alignment problem.

attached to the root node, the middle node, and the solution node. Names appearing black show that the person has been assigned a position. An arrow between names represents a fulfilled preference, whereas the circled number above a name yields the number of non-fulfilled preferences of that person.

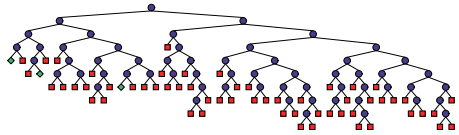


Fig. 5. Search tree for the photo problem.

An optimal solution of the problem is obtained by invoking all solution search in the Solver. The search tree looks as shown in Figure 5. Even though we are dealing with a rather simple example, it is already hard to even find the solution nodes in the tree. For this purpose, the Solver provides the possibility to hide all subtrees which contain only failed leaves. Invoking this functionality, the Solver displays a search tree as shown in Figure 6.

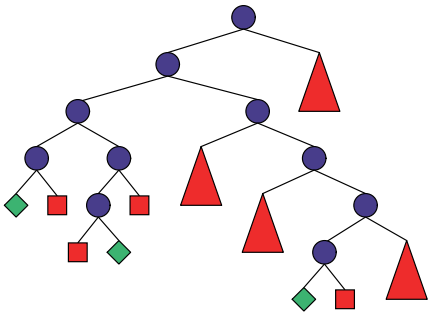


Fig. 6. Search tree with hidden failed subtrees for the photo problem.

By double-clicking the rightmost solution (the Solver assists in finding certain nodes by providing functionality to move a cursor to it) we get the optimal solution as shown in Figure 7.


Fig. 7. An optimal solution for the photo problem.

The Solver reports in its status bar that the entire search tree (as shown in Figure 5) has 59 choice nodes, 3 solution nodes, and 57 failed nodes. Compared to the size of our problem, this is a large number of nodes. Looking at the graphics of the first and the optimal solution it is clear that in order to keep the search tree small it is best to make choices such that they influence the satisfaction most. Thus we try a new heuristic from the finite domain library, by replacing `FD.enum.naive` in the program shown in Figure 2 by:

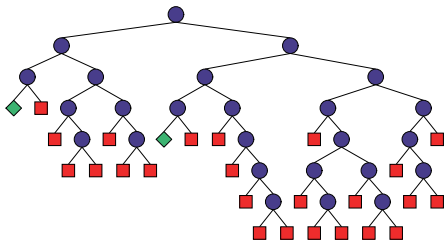


Fig. 8. Search tree for the photo problem with symmetries removed.

```
{FD.enum.generic Pos det constraints id min}
```

This selects the variable for enumeration with the most propagators depending on it. After applying the Solver to the modified search problem and the same order as before and invoking all solution search, the Solver gives in its status bar that the tree has now 52 choice nodes, 4 solution nodes, and 49 failed nodes. Thus the size of the tree has decreased by only about 10%.

If we have a second look at the search tree, we can see that it is much harder to prove the optimality of the last solution found, than to find this solution. Looking at the graphical representation of the optimal solution as shown in Figure 7, we see that for each solution there is a symmetric solution, which can be obtained by reversing the order of the people.

We want to reduce the search space by removing this symmetry. In our example it is fairly easy: We only require that two persons, say alice and bert, are placed in a fixed order. Thus we add the following constraint to our program:

```
Pos.alice >: Pos.bert
```

Applying the Solver to the new problem and searching for all solutions draws a search tree as shown in Figure 8. The tree now has only 25 choice nodes, 2 solution nodes, and 24 failure nodes. Thus, removing the symmetry reduces the size of the search tree by 50%.

## 5. The Solver

The Solver is invoked passing it the problem to be solved. Its usage does not require any change of the problem's definition: Instead of applying one of the predefined search strategies from the standard modules [3], the Solver can be used.

Depending on whether the Solver is applied to a problem or to both a problem and an order, it employs either normal search or best solution search for solving the problem. In both cases, the user can ask for the next or for all solutions. Prompting for the next solution creates the part of the search tree following a left-most depth-first strategy until a solution node is reached. Prompting for all solutions creates the entire search tree.

During the search process the user is kept informed by a status bar on how many nodes have been created and which depth has been reached so far. The search process can be interrupted and resumed by the user. The Solver

can be configured to halt automatically if it reaches a limit on the number of nodes in the search tree, or if it reaches a certain depth.

Search can be used in a completely interactive fashion: the user can create any part of the search tree in a step by step manner. Thus, promising paths in the search tree can be followed without being forced to create potentially huge but useless parts of it as would be necessary with a fixed strategy.

After the creation process of the search tree is halted, the Solver computes a layout for the newly created part of the search tree and updates the drawing of the tree. The drawn tree can be scaled by direct manipulation of a scale bar. Any subtree of the search tree can be hidden by replacing it by a small triangle. As already mentioned in the previous section, special support is provided to hide subtrees which contain only failed leaves. By visualising the search tree one can gain first insights into the search process: whether a first solution is found without too many failed nodes, whether it is hard to prove the optimality of the last solution found, and so on. The possibility of hiding failed parts of the search tree makes it easy to find the relevant paths leading to a solution.

All operations on nodes and subtrees are controlled by a cursor, which can be placed at any node in the already created part of the tree. Invoking an operation like hiding a subtree means to apply the operation to the subtree below the node selected by the cursor. The cursor can be positioned either by clicking the mouse to a node, or by navigation operations such as moving to the tree's root, to the next or previous solution, or to the previous not completely finished choice (so-called backtracking). Operations can be invoked by selecting a menu entry or pressing a key. Common operations are supported by buttons in an optional tool bar.

Each solved node carries as information its solution (a unary procedure). A choice node carries information on the constraints and propagators just before splitting of the choice point. The information is computed during the search process and is stored as a unary procedure containing the reflected computation space before choice splitting. The next section will explain this in more detail. Each node's information can be displayed in the Browser or in the Oz Programming Interface. It is also possible to compare the information attached to any two nodes, which assists in understanding what are the changes in the constraints between two nodes.

In the previous section it was shown that the Solver can be configured such that it employs a user-defined procedure to display the information. Of course, the same holds true for the comparison of nodes. It is of great importance that displaying and comparing information can be user defined. The information is understood best by tools which display the information in a form suited for the application. For instance, during the development of a scheduling program one wants to use Gantt-charts to display the information. Trying to understand what is going on in a scheduling application from the Browser's output alone is nearly impossible.

Besides the statistical information the Solver provides in its status bar, it is possible to display statistical information for each subtree. In the same way as described above, user defined procedures can be used to process and display the statistical information. For instance, a bar chart showing how many failures occur between solutions helps to understand how hard it is to prove optimality in best solution search.

## 6. Implementing the Solver

The Solver is written entirely in Oz. Even the part which deals with the creation of the search tree does not need any additional primitives other than the solve combinator (see Section 7). By having the primitives of the Oz Kernel language at hand, one has the expressivity to write useful programming abstractions. But even more, making use of these primitives one can also construct tools which can be used to debug the programming abstractions.

The Solver's implementation is around 6500 lines of Oz code. It can be divided into three parts:

1. Construction of the search tree. This part uses around 20% of the code.
2. Laying out and drawing the tree. This includes support for scaling the tree as well as support for hiding and unhiding of subtrees. These routines are approximately 15% of the entire code.
3. The user interface. This part provides for menus, the cursor management, management of dialogs, the status bar, and the tool bar. With 65% this part is by far the largest part of the Solver.

Taking the complexity of the code involved in the three parts, the first part is by far the most complex one. In the following two sections we will detail the implementation of the first two parts. The last part is the least complex. But, to make the tool work nicely and easy to use one needs to take care of a lot of small details, which explain the high percentage of code involved.

## 7. Creation of the Search Tree

In contrast to Prolog, Oz does not use a backtracking strategy to implement search. Instead, Oz provides for search by means of local computation spaces, in which speculative computation with constraints and propagators is performed. This encapsulates search, making it possible to have both speculative computations and computations interacting with external programs, e.g. the window system, at the same time.

All search functionality in Oz is obtained from a single primitive, called `SolveCombinator`. This is in contrast to languages like Prolog, where search is provided as a built-in concept, preventing the construction of user-defined search strategies. A detailed description of the `SolveCombinator` can be found in [10], [12].

As we have seen already in Section 4, a search problem in Oz is written as a unary procedure

```
proc {Problem Sol}
   Specification of the problem
```

```
end
```

where `Sol` is the so-called solution variable. It is constrained during the search process and eventually gives access to the solution computed.

The `SolveCombinator` takes the unary procedure `Problem` as input. Upon its application to `Problem` it creates a local computation space in which `Problem` is applied to a newly created solution variable. Reducing the application creates constraints and propagators depending on `Problem`. Now constraint propagation takes place. If during constraint propagation failure occurs the solve combinator will report this by constraining its single output variable to `failed`. After constraint propagation reaches a stable state, the combinator reduces.

In case that no choices are left in the computation space, the combinator yields

```
solved(P)
```

as output. `P` is a unary procedure containing the local computation space in abstracted form. Applying `P` gives access to the solution variable.

In case there are choices left, the choice is split up into two alternatives. The solve combinator gives these alternatives back as first-class citizens:

```
distributed(P1 P2)
```

`P1` and `P2` are unary procedures obtained by copying the local computation space and reflecting the copies into procedures.

The search tree constructed by the Solver is implemented as a tree of objects. The objects are instances of classes. For each type of node (i.e., choice, solved, and failed) there is a different class. The classes are created by multiple inheritance from classes which provide the functionality used for search tree construction, for computation of the layout, and for drawing.

```
fun {CreateTree P}
   case {SolveCombinator P}
   of failed then
      {New FailedClass init}
   [] solved(P) then
      {New SolvedClass init}
   [] distributed(P1 P2) then
      {New ChoiceClass
           init(sons: [{CreateTree P1}
                       {CreateTree P2}])}
   end
end
```

Fig. 9. Construction of the search tree.

The construction of the tree is controlled by the output of the solve combinator. That is, if the solve combinator yields `failed` as output, an object for a failed node is created. The same holds true for solved nodes and choice nodes. Choice nodes are created if the solve combinator yields `distributed(P1 P2)` as output. In this case, creation of the objects of search tree continues by creating the search subtrees for `P1` and `P2`. This is sketched by the function `CreateTree` as shown in Figure 9 which takes a procedure `P` as input and yields a tree of objects as output. Note that the creation of the search tree can be stopped and resumed at any place in the tree.

Besides of computing the search tree, also information which is attached to its non-failed nodes is computed. For solved nodes this is straightforward: A node's object stores the procedure containing the solution as computed by the solve combinator. For choice nodes this is more difficult: the information we want to have is the abstracted computation space when it has reached a stable state but before the choice is split into its alternatives.

The idea how to compute the information for the choice nodes is to introduce an additional choice point, where it has to be made sure that this choice point will be split first. When this choice point is split into two alternatives, both of them contain an abstracted computation space equivalent to the space just before attempting to split the original choice. One of the two obtained alternatives can be used to proceed with the creation of the search tree. The other alternative can be stored as information in the choice node's object.

The Solver demonstrates nicely how powerful search is in Oz. It exploits that search in Oz does not use backtracking: the copying approach allows to create and keep the full search tree augmented with information if desired. A backtracking approach would allow to keep information only on a single path from the root of the tree to some node. Moreover, it makes use of the fact that search is encapsulated: while constructing the search tree by doing speculative computation, the Solver communicates with the window system to control the graphical user interface. That the search process can be pursued interactively, would be impossible if a single fixed search strategy would have been built into Oz.

## 8. Laying Out and Drawing the Search Tree

After the search tree is created as described in the previous section, a layout for it is computed. Computing a layout for the tree means to compute coordinates for each of the nodes, such that drawing the nodes at these coordinates gives a pleasing graphical rendering of the tree.

Producing nice drawings of trees, or more generally, of graphs, is supported by various tools, e.g., VCG [9] and daVinci [2]. However, using one of them for the Solver is not appropriate for the following reasons:

- Most of the functionality to draw complicated, even cyclic, graphs provided by the tools is useless for our purpose.
- It is difficult to connect them to an application. Some of them provide only slow or even no support for incremental updates of the tree to be drawn.
- As suggested by the percentage figures shown in Section 6, the main issue in implementing the Solver is to tightly integrate the interactive search process with a nice graphical user interface. The tools are designed to be the "master" in an application, not just only a

component which can be incorporated and controlled easily within your own program.

In the literature there are several algorithms for producing nice layouts of trees. The Solver adapts an algorithm from [5] and extends it by making it incremental.

The idea of computing a layout is quite simple. The first thing is that the vertical coordinates are given by the depth of the node directly. Computing the horizontal coordinates is done as follows. Starting from the leaves of the tree one computes for each subtree its extent, that is the minimal borderline that encloses all of its nodes. Having computed the extents for the subtrees, one computes the minimal distance between two adjacent nodes such that there is some space left between their subtrees' extents. Then the subtrees' extents are merged together with the minimal distance computed before, yielding the extent for the parent. From the extents one can compute easily the horizontal position of a node relative to its parent node.

This algorithm can be made incremental by only using relative coordinates in the entire tree. This means that the position of a node is only given as offset to the parent's node position. This allows to adjust the layout of an entire subtree by just changing the offset to the subtree's parent. This property is also to be reflected by the drawing of the tree: it must be possible to move all nodes of a subtree by some horizontal offset.

## Acknowledgements

I would like to thank Gert Smolka, Peter Van Roy, and Jörg Würtz for being the first users of the Oz Solver. They provided useful tips and comments which helped to improve the Solver.

Tobias Müller, Joachim Niehren, and Peter Van Roy provided helpful comments on this paper.

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

## References

[1] Alain Bracquemond, Andreé Guilland, Michel Leconte, and Astrid Rios. CHARME user's guide. Technical report, BULL CEDIAG, 1995.

[2] Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph visualization system daVinci. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94*, volume 894 of *Lecture Notes in Computer Science*, Princeton, USA, 1995. Springer Verlag.

[3] Martin Henz, Martin Müller, Christian Schulte, and Jörg Würtz. The Oz standard modules. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

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

[5] Andrew J. Kennedy. Functional pearls: Drawing trees. *Journal of Functional Programming*, 1995. To appear.

[6] Micha Meier. Debugging constraint programs. In Ugo Montanari and Francesca Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 204–221, Cassis, France, September 1995. Springer Verlag.

[7] Tobias Müller, Konstantin Popov, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[8] Konstantin Popov. An exercise in concurrent object-oriented programming: The Oz Browser. In *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 29 November–1 December 1995.

[9] Georg Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205, Princeton, USA, 1995. Springer Verlag.

[10] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, 13–17 November 1994. The MIT Press.

[11] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[12] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, 1995.

[13] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1995. to appear.

[14] Gert Smolka and Ralf Treinen, editors. *DFKI Oz Documentation Series*. German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.