# CLPGUI: a Generic Graphical User Interface for Constraint Logic Programming

François Fages, Sylvain Soliman and Rémi Coolen
*(*francois.fages@inria.fr, sylvain.soliman@inria.fr*)*
*Projet Contraintes, INRIA-Rocquencourt,*
*BP105, 78153 Le Chesnay Cedex, France*

**Abstract.** CLPGUI is a generic graphical user interface for visualizing and controlling the execution of constraint logic programs. CLPGUI has been designed to be used in different contexts: initially for teaching purposes, then for debugging complex programs of real-world scale, and recently for developing end-user interfaces. The challenge of developing a tool which is generic w.r.t. both the constraint logic programming system and the visualizers, is addressed by a client-server architecture for connecting a CLP process to a Java-based GUI process, and by a non-intrusive tracing and control method based on annotations in the CLP program. Arbitrary constraints and goals can be posted incrementally from the GUI in an interactive manner, and arbitrary states can be recomputed. We describe several generic 2D and 3D viewers of the variables and of the search tree, and argue that the 3D representation is best-suited to apprehend the shape of large search trees. We also illustrate the use of CLPGUI for developing application-oriented end-user interfaces on two placement problems, one in virtual reality.

## 1. Introduction

Several tools for visualizing the execution of constraint programs have been developed in the last few years. These tools have been found very useful for debugging and improving constraint programs, and for teaching constraint programming. One can distinguish:

- *post-mortem visualization* tools, these tools are used after execution of the program, the program is annotated with specifications of the information to trace. This approach is implemented for example in the CHIP or CIAO systems, it allows using a wide variety of viewers, including both application oriented tools [24], and generic tools, for visualizing the search tree [5, 22], finite domain variables [4], or constraint propagation [23].

- *dynamic visualization* tools, these tools are connected to the constraint programming interpreter and realize an on-line visualization, possibly with animations [16]. This approach is implemented in the Grace tool [18] for finite domains visualization, and in OPL studio [2] for search tree and constraint propagation visualization.

Some of the tools in this category rely on the enrichment of the constraint solver to help improve the visualization, whether by grouping constraints [15] or by keeping explanations of domain reductions [13].

– *dynamic visualization and control* tools which allow interaction with a CLP process through different visualizations. One example is the Oz-Explorer system [21] where it is possible to jump to any previously encountered state by simply clicking on a node of the search tree, and restart computation from that state. User-guided search is implemented in Oz-Explorer using the first-class computation spaces of Oz. Recomputation is used to trade space for time in Oz-Explorer, and similarly in OPL studio [2], the state restoration mechanisms in tree search are described in [6].

In this paper we propose to push forward these ideas towards a generic architecture allowing the connection of a CLP process to dynamic visualization and control tools. Our ambition is not to realize an *ad hoc* tool limited to a particular constraint programming system, currently GNU-Prolog [10] and SICStus-Prolog [26], but a *generic* tool which can be ported to other constraint programming systems as well. One reason for this is that a wide variety of viewers can be useful for debugging or interacting with constraint programs and it is possible in this way to share developments. Another reason comes from the necessity to try different models which often imposes to change of constraint programming systems in order to benefit, for instance, from a particular global constraint.

Our approach relies in part on the generic trace format which is defined in the OADymPPaC consortium [19] for post-mortem analysis. We propose to extend this format with a similar generic format for control, and to use these formats for connecting on-line the CLP process to the GUI process. Our current implementation of CLPGUI in Prolog and Java is depicted in Figure 1.

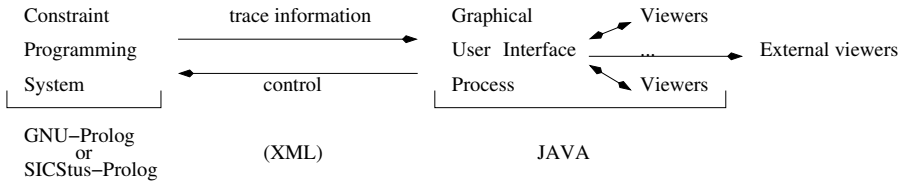


*Figure 1.* Information flow for dynamic visualization in CLPGUI.

Currently, most constraint programming systems do not support however the OADymPPaC trace format and the objective of gener-

icity of CLPGUI with respect to the constraint programming system is therefore quite challenging. The solution used in CLPGUI relies on a non-intrusive tracing and control method based on annotations in the program. Annotation predicates are defined for associating external names to variables, customizing the GUI and more importantly tracing the execution of the program. The originality of this approach lies in the use of annotation predicates not only for sending the specified trace of the execution but also, in the reverse direction, for interpreting control commands, such as recomputing a complete state corresponding to a traced node in the visualized search tree. We show that the annotation predicates form indeed a complete set of control points in the program which can be used to implement in a non-intrusive manner control commands such as backtracking or jumping to a different state represented as a node in the search tree.

Besides performance issues which in CLPGUI are largely solved, in order to scale-up, visualization tools have to rely on visualization paradigms that are still effective on large data sets. This difficulty is particularly severe for visualizing very large search trees which is a common need in constraint programming. Apart from [1], the visualization of search trees in three dimensions has not been much investigated. In this paper, we propose a 3D representation of search trees which we found most appropriate to apprehend the shape of large search trees, of $10^4$ nodes for instance. Playing with rotations, that cannot be reproduced in this article, is an important feature of the 3D representation which compensates the compactness of its 2D projection on the screen.

Such a robust architecture for visualizing and controling the execution of constraint logic programs can also be exploited to develop application-oriented end-user interfaces in both directions of visualization of solutions and control of the search. We exemplify this aspect of CLPGUI on two placement problems where the user can not only visualize the computed solutions but also modify the constraints or the solutions and search for new solutions near the modified ones.

The rest of the paper is organized as follows. Section 2 describes the client-server architecture of CLPGUI and the user console from which the execution of the CLP program can be controlled. Section 3 describes the annotation predicates used for tracing and controlling the execution of the CLP program at various levels of granularity. The interactive execution model of CLPGUI is also described with its implementation in two constraint programming systems: GNU-Prolog and SICStus Prolog with CLP(FD,R) libraries. Section 4.1 presents a dynamic 3D viewer for visualizing the evolution of the domain of variables over time. Section 4.2 describes the representation of the explored search space as partial CSLD derivation trees, and presents different visualizations with 2D

and 3D viewers. The following subsection provides some performance figures on a branch and bound optimization problem. Section 5 details the communication mechanism by message passing. Then in Section 6 we show how some application-oriented graphical user-interface have also been successfully developed with CLPGUI on two placement problems, one in virtual reality. Finally, we conclude on the generality of this scheme.

## 2. Client-Server Architecture

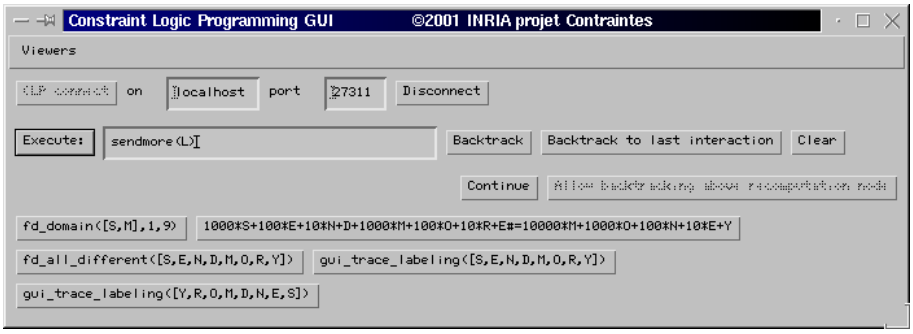


*Figure 2.* The CLPGUI console.

The graphical user interface of CLPGUI is a Java application connected by sockets as a client to a server which executes CLP goals. Both processes can run on different machines and communicate over the network. This has been experienced with CLPGUI for visualizing the execution of CLP programs on a Workbench of Virtual Reality. The choice of the Java language for implementing the GUI is motivated by several reasons:

- its object-orientation, all 3D viewers presented in the following sections inherit from a single class for moving and projecting 3D figures;
- the encapsulation of events handling, that is preponderant in dynamic visualization;
- the threaded execution, which is mandatory for implementing communication with the CLP process;
- its wide availability.

For efficiency reasons, we did not use the GL4Java or Java-3D libraries for the generic viewers presented in this paper, as they can directly benefit from *ad hoc* optimizations that speed-up their incremental display. Nevertheless the architecture can support the use of these powerful libraries for developing complex application-oriented viewers, as shown in section 6.2.

During initialization, the CLP server starts an interpreter of the command lines received on the socket. The GUI Java client opens a graphical console such as the one in Figure 2. That console is used for establishing a connection to the CLP server, and for posting constraints or executing arbitrary CLP goals.

The CLP program may contain annotations for creating buttons for some constraints or for some Prolog goals to execute in an interactive manner. These buttons for posting constraints or Prolog goals then appear at the bottom in the CLPGUI console, see Figure 2. Since these buttons rely on the annotation predicates discussed in the next section, they are completely independent of the underlying constraint system. A click on the button posts the constraint or executes the goal associated to the button. Other arbitrary goals can be executed by entering them in a text field. In addition, one button called "backtrack" continues the execution of the current goal up to the next success, or, if there is no more success, returns to the state of the previous interaction. Another button called "backtrack to last interaction" forces backtracking to the state of the previous interaction. The menu bar of this console contains menus to select and activate the viewers of the search tree or of the finite domain variables.

## 3. Non-Intrusive Traces and Control through Annotations

Annotations have been proposed as a simple mechanism for tracing the execution of constraint logic programs and specifying the level of granularity of the data to visualize [5]. In this section, we present the main annotation predicates defined in CLPGUI and we show how annotation predicates can be extended to an active mechanism for interpreting control commands as well in a non-intrusive manner.

### 3.1. Annotation Predicates

In CLPGUI, the CLP program may contain annotations for giving an external name to CLP(FD) variables, for creating buttons for posting constraints or goals from the CLPGUI console, and for specifying the goals to visualize in the search tree. The following predicates are part of the annotation library:

- `gui_varnames(LV,LN)` and `gui_varnames(LV)` give an external name to the list `LV` of CLP variables. These external names are used in the graphical user interface and for the communication by sockets. If no names are provided, standard names `V1`, `V2`, ... are created.

- `gui_button(goal)` creates a button in the GUI console for executing a goal or for posting a constraint.

- `gui_bagof_buttons(goal, call)` creates a bag of buttons for each successful instance of the second argument.

- `gui_trace_search(goal)` executes the goal and traces the execution of that goal, by creating nodes in the search tree. The goals and constraints posted from the graphical console are always traced.

- `gui_show_domains` updates the visualization of the current state of FD variables.

The advantages of annotations are:

- the flexibility of defining different levels of granularity concerning the information to visualize,

- the easiness for making existing programs interactive,

- the portability of the GUI to other constraint programming systems, as all communications with the GUI are encapsulated in the implementation of annotation predicates.

The limitations of annotations are well-known in standard programming environments: they may be difficult to maintain in large programs. In that case, one solution is to automatically generate annotations with a graphical editor of the program source, where spy points and trace options can be specified. Nevertheless, one peculiarity of constraint logic programming is the conciseness of programs. CLP(FD) programs for solving combinatorial optimization problems on real-size data may compute with a huge amount of constraints and variables, but the program source for handling constraints and defining complex search strategies usually remains relatively concise. Therefore in this context, the proposed annotations appear as a satisfactory solution.

In many CLP systems however, the heuristic labeling procedures are built-in, and may be difficult to trace precisely with simple annotations. This is possible if the CLP system provides coroutining facilities like for instance the `freeze` predicate) of SICStus-Prolog [26]. In this case,

one can trace the instantiation of variables and the search tree created by the built-in labeling procedure, with a simple call to the following predicate before labeling:

```
freeze_trace([]).
freeze_trace([X|L]):- freeze(X,gui_trace_call(X=X)),
                      freeze_trace(L).
```

In absence of coroutining predicate, one solution is to rely on the tracing facilities of the CLP system in order to extract, and communicate to the GUI, the relevant information, like the creation of a choice point or the reduction of one variable's domain. Another solution is to program the labeling heuristics in the host language, in order to make available the information coming from the constraint solvers that is relevant to the search heuristics. In that case, the effect of the search strategy can be visualized at different levels of granularity. In its simplest form, a predicate for tracing a labeling procedure can be defined with a `gui_trace_search` annotation as follows:

```
gui_trace_labeling([]).
gui_trace_labeling([X|L]):- gui_trace_labeling(L),
                 gui_trace_search(fd_labeling(X)).
```

It is worth noting that if the search strategy is implemented with a meta-interpreter, and uses constraint posting instead of labeling (like in the bridge problem described in section 4.3), the relevant part of the search tree can always be traced with CLPGUI annotations.

A similar difficulty arises for tracing internal constraint propagation steps. This is not possible without access to the wakening events of the constraint solver, as defined for instance in the OADymPPaC format [19]. The annotations for tracing constraint propagation steps have thus to rely either on coroutines or on the tracing facilities of the solver in order to extract and communicate constraint wakening events. An XML syntax for traces has been defined by the OADymPPaC consortium for this purpose.

*Example 1.* The following annotated GNU-Prolog program solves the well known SEND+MORE=MONEY puzzle in an interactive manner, by creating buttons for posting the constraints and for trying two labeling goals in this example:

```
sendmore(L):-
        L=[S,E,N,D,M,O,R,Y],
        gui_varnames(L,['S','E','N','D','M','O','R','Y']),
        fd_domain(L,0,9),
        gui_show_values,
```

```
gui_button(fd_domain([S,M],1,9)),
gui_button(1000*S+100*E+10*N+D+1000*M+100*O+10*R+E
           #= 10000*M+1000*O+100*N+10*E+Y),
gui_button(fd_all_different(L)),
gui_button(gui_trace_labeling(L)),
reverse(L,L2),
gui_button(gui_trace_labeling(L2)).
```

This program generates the console in Figure 2. The evolution of the finite domain variables over time, after the posting of constraints and of the first labeling goal, is depicted in Figure 5. The visualization of the search tree for obtaining all solutions under the first labeling goal, and then under the second labeling goal executed after a backtracking command, is depicted in Figure 3. Under the first ordering, the labeling is deterministic. Under the second ordering, few backtracking steps occur on variable `Y` when searching for other solutions. Note that other labeling heuristics can be tried directly from the console. On such pedagogical examples, the advantage of immediately visualizing the effect of posting a constraint or trying a labeling, is clear for teaching purposes, since the user can see at the same time the number of backtracks, the depth of the search, and the evolution of the domains of the variables.

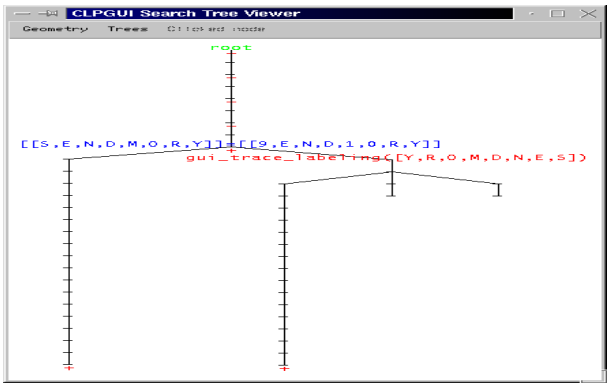


*Figure 3.* Search trees with two labeling orderings in the puzzle SEND+MORE=MONEY.

*Example 2.* The following program solves the N queens problem by creating buttons for posting the constraints (`safe` predicate) and labeling goals for each variable (`fd_labeling` predicate) and for all variables (`gui_trace_labeling` predicate).

```
queens(N,L):-
```

```
length(L,N),
fd_domain(L,1,N),
gui_show_values,
gui_button(safe(L)),
gui_bagof_buttons(fd_labeling(X),member(X,L)),
gui_button(gui_trace_labeling(L)).
```

Three visualizations of the search tree for one single execution of the above program for the 8 queens problem are depicted in Figures 6, 7 and 8.

### 3.2. Control through Annotations

Annotations in the program provide a set of control points which can also be used to implement a sophisticated control of the execution of the constraint program in a non-intrusive manner.

In CLPGUI we use annotation predicates to implement the automatic recomputation of any state represented as a node in the search tree. Once a node in the search tree (such as the tree depicted in Figure 3) has been selected for being recomputed (by clicking on it), a recomputation directive is transmitted to the solver by means of the path in the tree from the root to the node. This path is then interpreted as a sequence of choices that have to be taken from the initial state to the one being recomputed. The algorithm is thus basically the classical recomputation algorithm of a path in a derivation tree [21, 2, 6], where actually no intermediate state is memorized. The novelty in CLPGUI is that this recomputation algorithm is implemented in a non-intrusive manner using annotation predicates to control the execution of the CLP process following the recomputation path.

This recomputation algorithm supports CLP programs containing cuts and exceptions. It does not support however programs containing side effects. One way to support side effects would be to recompute not only the path but the entire derivation tree leading to the recomputed node. Another way would be to memorize the intermediate state values that are subject to change by side effects. A particular case of this is the memorization of the best cost value in optimization predicates, such as in the CLP predicate `minimize(Goal,Cost)` for instance.

Batch recomputation, i.e. memorizing all constraints posted in a node and adding all constraints at once for a recomputation, has been proposed in [6] as a method giving slightly better performances. It is worth noting however that this method is not applicable in our non-intrusive setting as we do not assume a complete knowledge of all constraints posted by the program.

### 3.3. Interactive Execution Model for CLP

The interactive execution model of the CLP process used in CLPGUI is a combination of the model for adding and removing constraints and goals described in [12] with the non-intrusive recomputation algorithm described above. In CLPGUI, constraints and goals can only be added to the current goal; the removing of constraints or goals occurs by backtracking or jumping to a specific node. It is therefore possible on a success of the current goal:

- to add constraints or any goals to the current goal and continue resolution,
- to backtrack to the next success (command "backtrack" of Section 2),
- to backtrack to the previous interaction (command "backtrack to last interaction"),
- or to recompute a given state depicted as a node of the search tree.

It is worth noting that such a top level is in fact very appropriate for standard Prolog systems. Our current implementation uses the global variables of GNU-Prolog [10] to memorize global information, such as input and output sockets, variable names, and information used for backtracking. Global variables make it possible to avoid adding parameters to many predicates and lead to a simple implementation of annotations. In SICStus-Prolog [26], global variables are emulated using blackboard predicates, mutables and system predicates.

## 4. Generic Viewers for Debugging or Teaching

### 4.1. 2D and 3D Views of Finite Domain Variables

The domains of finite domain variables at any given time can be visualized with a generic 2D boolean matrix having one row per variable and one column per value. In the N-queens problem this view provides a view of the chessboard. In a scheduling problem, like the famous bridge problem [27], this view shows the possible starting dates of the task and the flexibility of a solution, as depicted in Figure 4.

The evolution of finite domain variables over time can be visualized in a three dimensional graph variable-domain-time, as already proposed in the VIFID/TRIFID tool [25]. In CLPGUI the visualization is dynamic, the Java process reads the stream of finite domains information

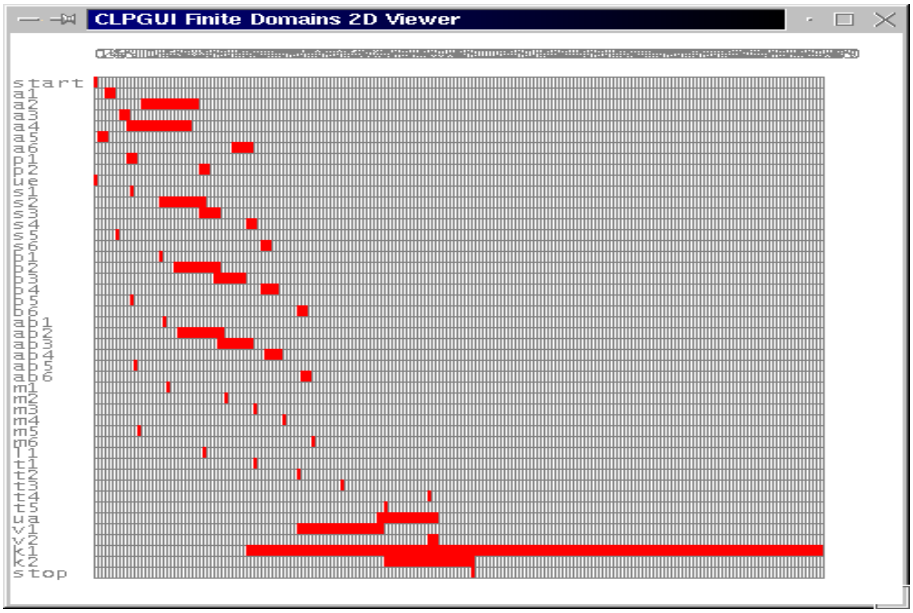


*Figure 4.* 2D view of finite domain variables in an optimal solution of the bridge scheduling problem [27].

and paints the figure in an incremental manner. Domains are depicted by their size on the vertical axis, see Figure 5. According to options, that can be set in the CLP program or in the GUI, only the size, the interval or the complete domain of variables is visualized. But in any case only the sizes of the domains are memorized, therefore the extra information is lost when the figure is repainted. The time axis traces the interactions (i.e. the posting of constraints in the example), and the execution of traced goals (i.e. the labeling in the example). This view shows that the posting of constraints instantiate variables $S, M, O$ and that the first labeling step on variable `E` in fact instantiates all variables by constraint propagation. An option determines whether backtracked states are traced or erased.

The figure can be moved, zoomed and rotated. For efficiency reasons, the rotations are limited to a quadrant of a sphere which is not a real limitation for the user. In this way the visible faces are efficiently determined and the figure can be drawn incrementally.

Extra information on variables and executed goals can be obtained by moving the mouse over the position of a variable or on a time position.

The 3D dynamic view of finite domain variables evolution is very useful for teaching constraint programming. The effect of constraints

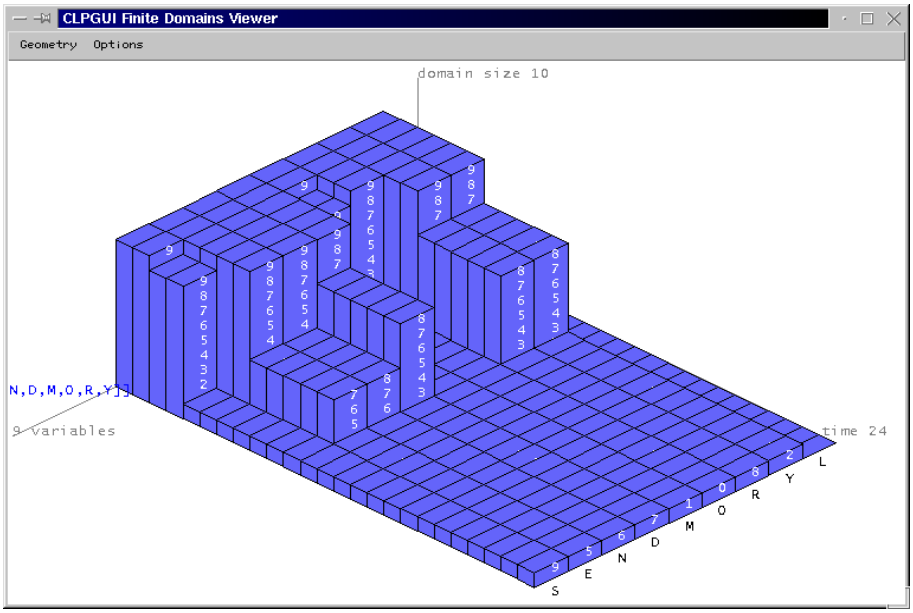


*Figure 5.* Dynamic 3D view of finite domain variables in the puzzle SEND+MORE=MONEY.

is immediately seen and many strategies can be tried step by step. The possibility for a student to *see* the domain reductions as they are happening is really helpful for understanding the underlying machinery of constraint programming. On larger sets of variables, the 3D view of domains can still be useful to get a view of the pruning power of different constraint models, and of the efficiency of different search heuristics, by comparing the general shape of domain reductions.

## 4.2. 2D and 3D Views of the Search Tree

### 4.2.1. *Partial CSLD derivation trees*

The search tree considered in CLPGUI is a labeled tree defined as follows:

- a node is introduced for each call to a traced goal (called a *call node*), and for each success to a traced goal (called a *success node*),
- the label of a call node is the called goal,
- the label of a success node is the list of named variables with their value,
- the arcs correspond to the operational CLP transitions.

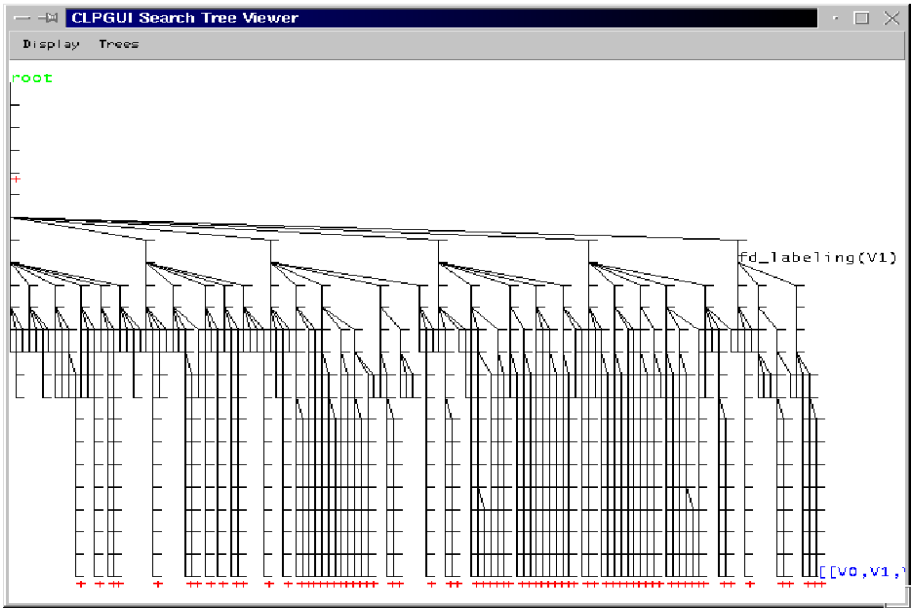


*Figure 6.* 2D view of the search tree in the 8-queens problem.

This tree, which is a subtree of the CSLD derivation tree [17], is a quite natural representation of the search tree for describing CLP program execution. A branch represents a conjunction, and the different successors of a node represents a disjunction. A success node may have several successors if there is an untraced non-deterministic goal which is executed after the success, and before the next call to a traced goal. This is the main reason why success nodes are introduced in partial CSLD trees. In this way, the non-determinism due to untraced goals cannot be confused with the non-determinism of traced goals.

One disadvantage of CSLD trees is that when dealing with deterministic programs they are threadlike and thus space consuming in their standard representation. AND-OR trees provide a more compact representation, as the threadlike parts of the CSLD tree are compacted in the successors of a single AND-node. For this reason, in the context of logic programs where most predicates are deterministic, AND-OR trees, and their variant AORTA diagrams which indicate the status of resolution of the goals, have been preferred [11]. Nevertheless in the context of constraint logic programming over finite domains, the situation is quite different. The search tree to visualize is usually focused on the labeling predicates, or more generally on the branching procedure, which is highly non-deterministic (at least during debugging). The representation of the deterministic part of the search tree with threadlike

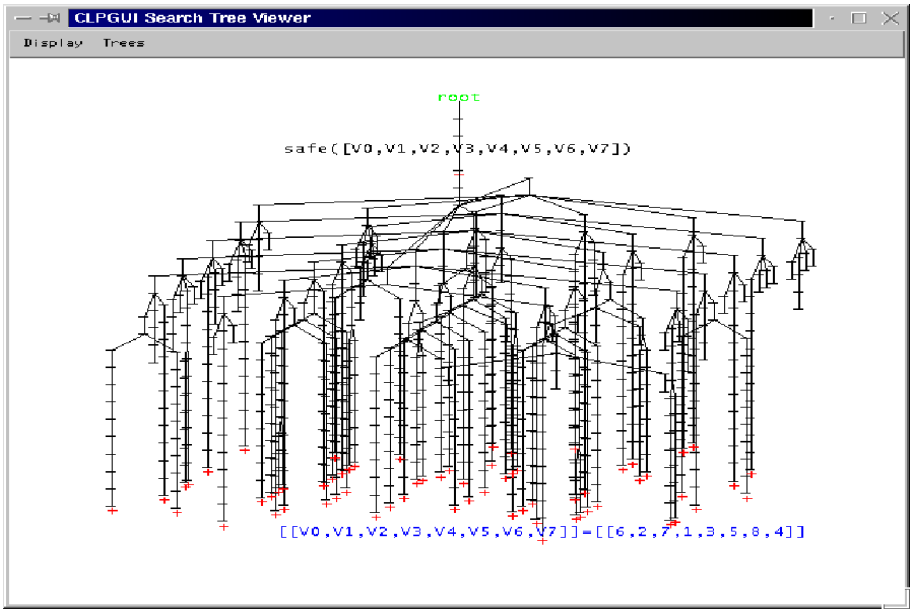


*Figure 7.* 3D of the search tree in the 8-queens problem.

structures provides an immediate visualization of the pruning power of constraints.

A naive solution for tracing constraint propagation steps in this approach is to add deterministic nodes for tracing constraint wakening events. For space limitation reasons, it is preferable however to aggregate constraint propagation information to the nodes of the search tree. This is proposed in the "Christmas trees" of OPL studio [2].

For search engines not based on backtracking, it is worth noting that a partial CSLD derivation tree can still provide a valid representation of the explored search space, as long as the explored states can be defined by their relation to some ancestor states. A formalization of an interactive constraint solver by transformations of CSLD derivation trees was done in [12].

#### 4.2.2. *Search Tree Viewers*

Once the search tree is formally defined, it can still be visualized in many ways, and in some cases it can be interesting to use several visualizations at the same time. We have currently implemented several two-dimensional and three-dimensional viewers, but many more representations could be imagined and fruitfully used.

In all the following representations, the labels of the nodes are visualized when the mouse is moved over them, and there exists an option

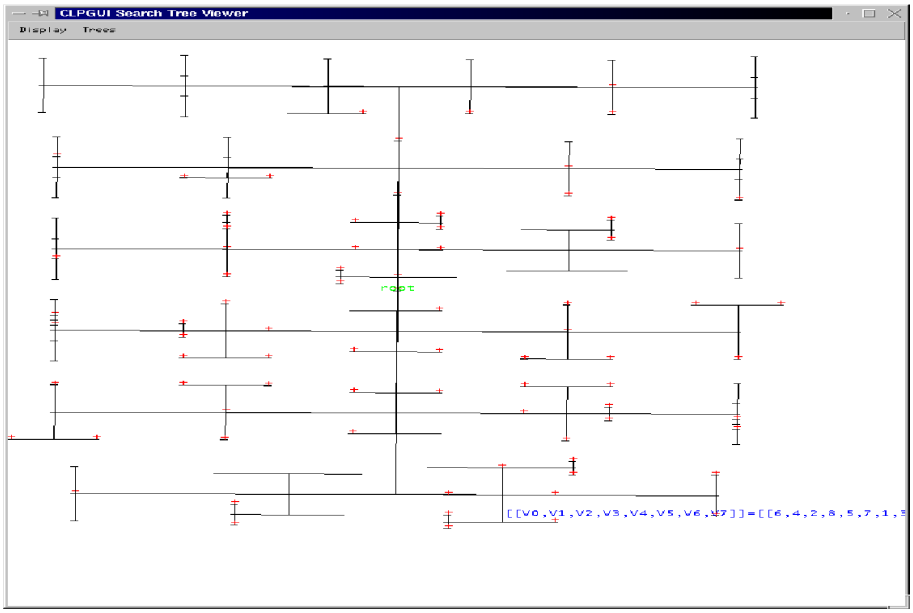


*Figure 8.* Dual treemap representation obtained by rotation of the 3D view.

for making all nodes visible. The successes are materialized by a red cross. Each view can be moved, zoomed and rotated.

Figure 3 uses a standard 2D representation of the search tree in a fixed width. Figure 6 uses a dynamic 2D representation of the tree with a fixed spacing between leaves. This representation of the tree can be drawn incrementally and is thus appropriate for the dynamic visualization of large trees.

To our knowledge, the 3D visualization of search trees has not been much investigated. Figure 7 shows a somewhat original 3D representation of the search tree with alternating planes of successors. One advantage of this 3D representation is that it is relatively compact, it helps visualizing rather large trees by playing with rotations, see Figure 11 for another example. Our experience is that the 3D view is the most appropriate view to apprehend the shape of large search trees.

It is interesting to note that one obtains a dual treemap representation of the tree by rotation of the 3D alternate tree up to its vertical projection, as done in Figure 8. Treemap representations (with colors for aggregating information) are known to be particularly efficient for representing large data sets [20] and for visualizing complex phenomenons such as correlations, patterns or symmetries. By rotating the 3D tree up to its vertical projection, one obtains a dual view of the treemap where the centers of the rectangles instead of the rectangles

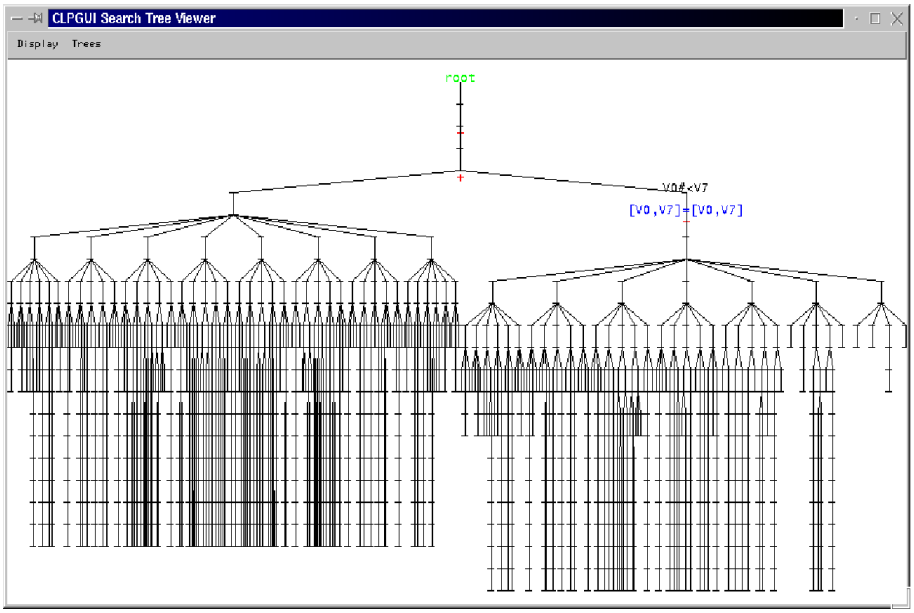


*Figure 9.* Complete search trees for all solutions to the 8 queens problem, without and with symmetry breaking obtained by adding the constraint `V0#<V7`.

are depicted. Figure 9 depicts the search trees for all solutions to the 8 queens problem obtained on the left without symmetry breaking, and on the right obtained by adding the constraint `V0#<V7`. Figure 10 depicts the same search trees as dual treemaps, which show the drastic improvement obtained by symmetry breaking, namely the lower number of choice points needed to find non-symmetric solutions.

In any of the 2D or 3D trees shown before, the user can select a node by clicking on it and then use the menu to hide/show the subtree below that node, or more importantly, to recompute the state of that node as current state, as already described in section 3.2. The ways in which the user can interact with the running CLP process are thus not limited to those provided by the user console of Figure 2 for interactive execution and user-guided backtracking. Jumps to arbitrary traced states can be specified directly from the search tree viewers.

### 4.3. Evaluation

Our experience of using CLPGUI for teaching constraint programming has been very positive. The dynamic visualization of CLP programs really speeds-up the process of learning the basic concepts of domain filtering, constraint propagation and search trees. Just giving a demon-

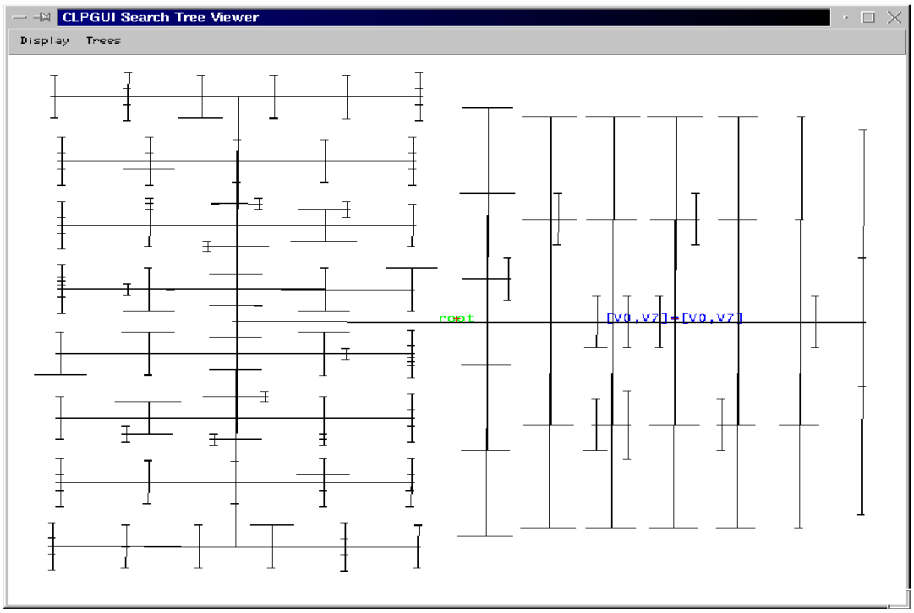


*Figure 10.* Complete dual treemaps for all solutions to the 8 queens problem, without and with symmetry breaking.

stration of CLPGUI in a course provides immediate intuition to students. CLPGUI has also been fruitfully used to visualize the search tree of CLP(R) programs. Concerning debugging, CLPGUI appears as a complementary tool to standard debuggers which may present more fine grained trace information. The main advantage of CLPGUI is to immediately apprehend the shape of the search tree and the shape of domain reductions and to visualize the effects of adding constraints or of changing the labeling procedure.

On real-size data, the visualization of the search tree in CLPGUI shows satisfactory performance figures. The drawing of the tree is immediate and the figure can be moved and rotated without difficulty on large examples. Figure 11 shows the search tree obtained with GNU-Prolog for the bridge problem [27], a medium size job-shop scheduling problem using optimization. The branch and bound procedure develops search trees in three parts. The first part corresponds to the enumeration of solutions with decreasing costs (for minimization problems). The second part exhausts the search space to show that there does not exist a better solution than the last solution found. The second part of the search tree constitutes the proof of optimality. The third part which is optional enumerates all optimal solutions by fixing the cost to its optimal value. The first descent (on the left) corresponds to the search

of the first solution of cost 110. It contains 78 call nodes. The second descent corresponds to the search of a better solution of cost 106. It contains 78 call nodes. The third descent contains some backtracked branches and corresponds to the search of the optimal solution of cost 104. It contains 99 call nodes. The fourth part of the tree is a dense subtree of 3728 call nodes which corresponds to the proof of optimality (it is a finitely failed subtree). The fifth branch enumerates all optimal solutions of cost 104. The 3D view is the most appropriate view for the tree of the proof of optimality. It can be moved and rotated without difficulty.

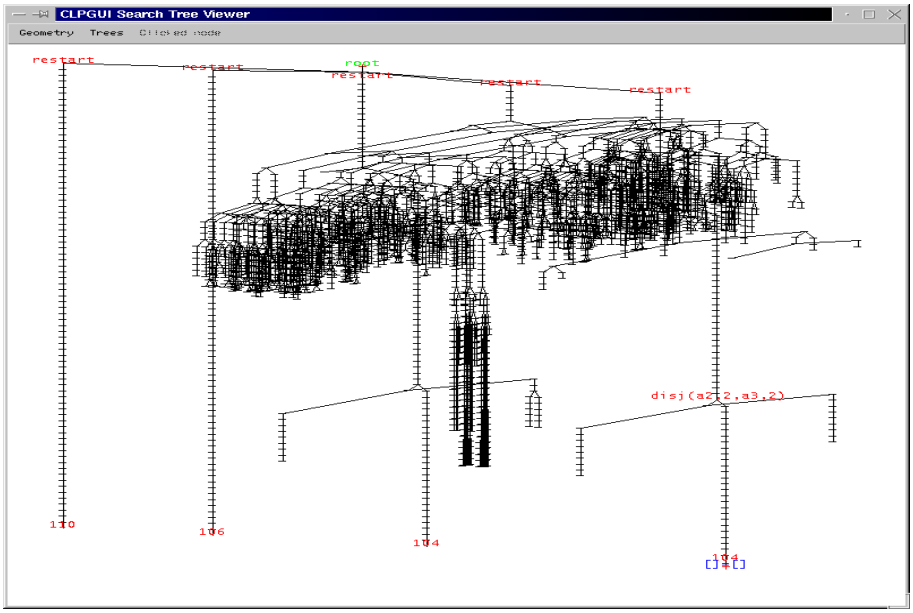


*Figure 11.* 3D view of the search tree for the bridge problem in GNU-Prolog (3905 call nodes).

We report here the timings obtained on a Pentium III 600 MHz processor under Linux. GNU-Prolog solves the bridge problem mentioned above in 100 ms, including the proof of optimality. The solving together with the visualization of the search tree with 3905 call nodes takes 470 ms with CLPGUI. This overhead is due to the communication of messages by sockets. The overhead was reduced from 2600 ms to 470 ms by optimizing socket calls.

## 5. Communication by Message Passing

In this section we describe the messages which are transmitted between the CLP process and the GUI process. The CLP process produces the trace information specified by the annotations in the CLP program, or asked from the GUI. The messages emitted from the CLP to the GUI are the following:

- `<variables ...>` sends the list of FD variable names
- `<button G>` asks the GUI to create a button for posting the constraint or goal `G`
- `<undo button G>` indicates backtracking on the creation of a button for `G`
- `<node G>` traces a call to goal `G`
- `<undo node G>` traces backtracking on the call to `G`
- `<child G>` traces a success to `G`
- `<undo child G>` traces backtracking on the success to `G`
- `<undo goal G>` indicates backtracking on the call to goal `G`
- `<domainSizes ...>` sends the domain sizes of FD variables
- `<domainIntervals ...>` sends the current intervals of FD variables
- `<domainValues ...>` sends the current finite domains of FD variables (represented as a list of intervals)
- `<undo domainValues>`, `<undo domainIntervals>`
  `<undo domainSizes>` warns the GUI that the finite domain variables are updated by backtracking.
- `<success>` indicates that the current derivation is a success
- `<clear>` indicates return to top level.

In the other direction, the messages emitted from the GUI to the CLP process are the following:

- `<showSize>`, `<showInterval>`, `<showValues>` sets the information on finite domains that need be sent
- `<execute G>` asks to post constraint `G` or execute goal `G`
- `<backtrack>` asks backtracking to the next success
- `<backtrackInteraction>` forces backtracking to the last interaction
- `<recompute ...>` asks for recomputation following the given path
- `<clear>` asks to abort the current execution.

The portability of CLPGUI to a new constraint programming system is determined by the ability of the constraint programming system to produce and interpret these communication messages. The messages of the first list are produced by the predicates of annotation library

described in Section 3. The messages in the second list are interpreted by the interactive execution model described in the previous section.

## 6. Application-oriented Graphical Interfaces

This section shows that it is also possible to develop end-user graphical interfaces customized for one application by taking two concrete examples in which new viewers were created and plugged into CLPGUI to allow for specialized views of the computed solutions but also for specific interactions with the solver.

### 6.1. A Bin Packing Problem with Rotations

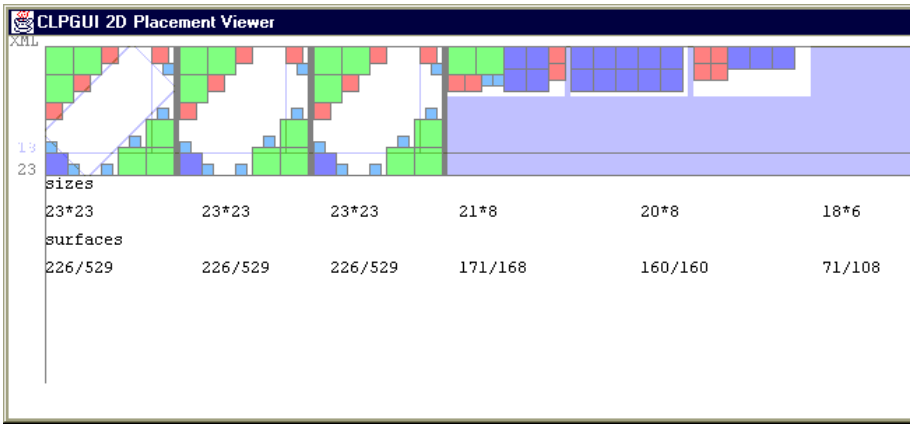


*Figure 12.* Automatic placement viewer for the UIST contest.

The second interface-design contest of UIST 2002 [3] provided a good test for the useability of CLPGUI in a challenging situation: a complex problem to be solved as efficiently as possible in a minimal time, all this relying a lot on the user interface.

The problem was to pack a set of squares of different sizes into three bins, the sizes of which having to be minimized, and rotations of squares being allowed. We developped two ad hoc graphical interfaces plugged into the CLPGUI architecture for this problem. The first was used to visualize an automatic placement of the squares, either a straight placement or a placement in the corners of the three bins, as shown in Figure 12, in order to insert the squares in the middle using rotations. An important feature of this interface was that the squares could be moved by drag and drop in order to make some manual modifications

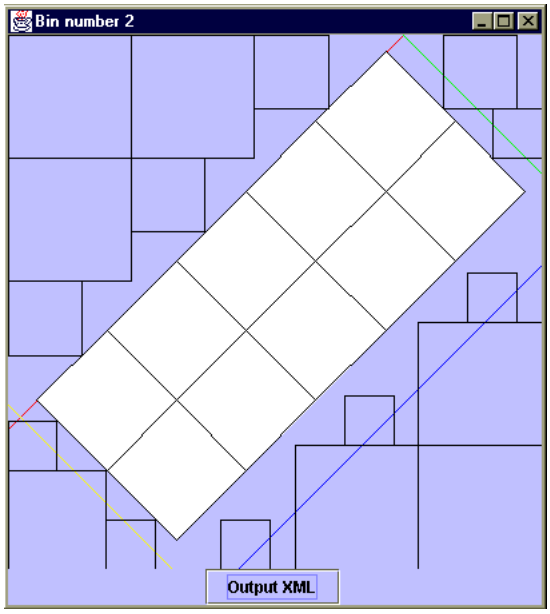


*Figure 13.* Rotation and square-squeezing viewer for the UIST contest.

of the initial placement solution. The second specific viewer, shown in Figure 13, was used to show the filling of the center of the bins with the remaining squares after squeezing to the optimal (non-integer) size.

All the parameters were modifiable from the GUI, all interactions were backtrackable and the recomputation capabilities of CLPGUI proved quite useful. The automatic placement module used SICStus Prolog's global constraints for non-overlapping rectangles [26]. A typical scenario was to use the automatic placement module for finding best straight placements, possibly with the help of manual interactions, and then to use it for finding alternative placements with rotations, again with manual interactions for trying different parameters or moving some squares manually. CLPGUI earned the award of "Best User Interface concept" in that challenge.

### 6.2. User-Interactions in Virtual Reality for a 3D Placement Problem

We have investigated the use of CLPGUI for combining a constraint programming approach to a 3D placement problem, namely placing furniture in an office, with virtual reality tools used for visualizing the solutions automatically computed by the system, manually modifying these solutions, recomputing new solutions near the modified configura-

*Figure 14.* Two different views of a placement which can be modified in virtual reality.

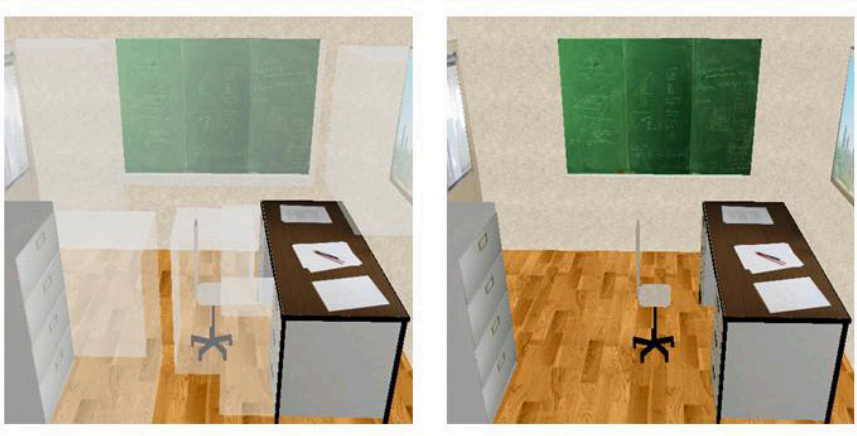

*Figure 15.* Visualization of distance constraints which can be modified graphically.

tion, and also for modifying the constraints of the problem [7]. The idea was to explore the concept of constraint programming as a paradigm of augmented virtual reality, where the augmentation comes from the visualization of the constraints.

The user interacts with the placement system through a representation of the scene in virtual reality, augmented with a representation of distance constraints. The application-oriented scenery viewer is implemented in GL4Java [14], for its perfect performances in rendering

textured 3D views. The viewer allows us to see the scenery from different angles (from above, or inside the room with a position and an angle chosen with the mouse), as shown in Figure 14. The viewer also allows the user to interact with the system by picking some furnitures to move them around, and recompute a feasible solution which minimizes the distance to the manually modified configuration. Information used in the solver, like distance constraints, can also be observed and modified through their visualization in augmented virtual reality, as shown in Figure 15.

On the CLP side, the constraint program developed in SICStus Prolog uses a global placement constraint for placing the objects without overlapping, plus some specific constraints of orientation, symmetry breaking techniques, different heuristics and optimization procedures.

*Figure 16.* A 3D scenery with 50 furnitures.

Some heuristics scale-up as shown in Figure 16 with a problem of 50 furnitures. However the emphasis of the system is on its interaction capabilities as it seems impossible to define general automatic placement strategies. Having CLPGUI as backbone for this application provides for free the ability to communicate easily between the graphical user-interface and the solver, and to develop very complex interactive solving strategies.

## 7. Conclusion

We have described a generic graphical user interface for visualizing and controlling the execution of constraint logic programs. The open archi-

tecture of CLPGUI involves a CLP process and a GUI process which communicate by sockets. This choice has proved efficient enough for the dynamic visualization and user-interactions on large examples. An important reduction of the overhead was obtained by optimizing socket calls and by using simple data compression techniques for communication. The tracing and control facilities are implemented in CLPGUI in a non-intrusive manner with annotations which allow the user to specify the required level of granularity. We have shown that the annotation predicates can be used also to implement non-intrusively the automatic recomputation of any state represented as a node in the search tree.

CLPGUI supports the use of different viewers, either generic or application-oriented. The 3D visualization of search trees described in the paper, is often the preferred view to apprehend the shape of large search trees by playing with rotations, as it is very compact. More work is needed however to parametrize the different viewers and invent novel visualization paradigms of complex data. In this respect the flexibility of the architecture makes it possible to connect CLPGUI to external generic viewers, or to use powerful libraries like GL4Java to develop application-oriented viewers as shown here with a placement problem in virtual reality.

The most obvious limitation of CLPGUI in its present state is the absence of trace for fine grained constraint propagation events. The generic trace format for finite domain constraint solvers defined by the OADymPPaC consortium [8, 19] is under implementation for GNU-Prolog and Choco, and will be used in the future versions of CLPGUI. One possibility is to visualize such fine grained data by aggregating them into a special representation of nodes in the search tree using different sizes and colors like in Christmas trees [2] or in conventional treemaps.

In another direction, CLPGUI is not a visual programming tool as far as the capabilities for defining new goals from the GUI are rudimentary. Nevertheless the architecture of CLPGUI could support visual programming techniques by adding the capability of defining new constraints and goals graphically, which is certainly worth investigating.

## Acknowledgements

We would like to thank all members of the OADymPPaC RNTL project of the French Ministry of Research, and especially Abderrahmane Aggoun, Thomas Baudel, Pierre Deransart, Jean-Daniel Fekete, Ludovic Langevine and Mohammad Gonhiem for interesting discussions on this topic. We are also grateful to Anupam Agarwal for optimizing com-

munication by sockets in CLPGUI, and to Jean-Michel Leconte for preliminary work along these lines on the workbench of virtual reality at INRIA.

## References

1. Bouvier, P.: 2000, 'Visual Tools to Debug Prolog IV programs'. in [9], Chapt. 6.
2. Bracchi, C., C. Gefflot, and F. Paulin: 2001, 'Combining Propagation Information and Search-Tree Visualization using OPL Studio'. In: A. J. Kusalik, M. Ducassé, and G. Puebla (eds.): *Proceedings of WLPE'01*. Cyprus, pp. 27–39.
3. Brady, T., J. Marks, and K. Ryall: 2002, 'The Second ACM UIST Interface-Design Contest'. 15th ACM Symposium on User Interface Software and Technology. http://www.acm.org/uist/uist2002/contest/.
4. Carro, M. and M. V. Hermenegildo: 2000a, 'Tools for Constraint Visualization: The VIFID/TRIFID Tool'. In: *[9]*. pp. 253–272.
5. Carro, M. and M. V. Hermenegildo: 2000b, 'Tools for Search Tree Visualization: The APT Tool'. In: *[9]*. pp. 237–252.
6. Choi, C. W., M. Henz, and K. B. Ng: 2001, 'Components for state restoration in tree search'. In: *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming* CP *'01)*. Cyprus, pp. 240–255.
7. Coolen, R.: 2003, 'Apport de la réalité virtuelle dans un problème de placement 2D/3D'. Rapport de stage d'option de l'ecole polytechnique, INRIA.
8. Deransart, P., M. Ducassé, and L. Langevine: 2002, 'A Generic Trace Model for Finite Domain Solvers'. In: B. O'Sullivan (ed.): *Proceedings of User Interaction in Constraint Satisfaction (UICS'02)*. Cornell University (USA), pp. 32–46.
9. Deransart, P., M. V. Hermenegildo, and J. Maluszynski (eds.): 2000, *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, Vol. 1870 of *Lecture Notes in Computer Science*. Springer-Verlag.
10. Diaz, D.: 1999–2003, 'GNU-Prolog user's manual'. http://gprolog.inria.fr.
11. Eisenstadt, M. and M. Brayshaw: 1988, 'The Transparent Prolog Machine: an execution model and graphical debugger for logic programming'. *Journal of Logic Programming* **5**(4), 277–342.
12. Fages, F., J. Fowler, and T. Sola: 1995, 'A Reactive Constraint Logic Programming Scheme'. In: L. Sterling (ed.): *Proc. International Conference on Logic Programming ICLP'95*. Tokyo.
13. Ghoniem, M., N. Jussien, and J.-D. Fekete: 2003, 'Visualizing explanations to exhibit dynamic structure in constraint problems'. In: B. O'Sullivan (ed.): *Proceedings of the Third International Workshop on User-Interaction in Constraint Satisfaction*. Kinsale, Ireland. http://www.cs.ucc.ie/ osullb/UICS-03/.
14. Goethel, S.: 1997–2003, 'OpenGL for Java online documentation'. http://gl4java.sourceforge.net/docs/.
15. Goualard, F. and F. Benhamou: 1999, 'A Visualization Tool for Constraint Program Debugging'. In: *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*. Cocoa Beach, Florida, pp. 110–117.
16. Goualard, F. and F. Benhamou: 2000, 'Debugging Constraint Programs by Store Inspection'. in [9], Chapt. 11.

17. Jaffar, J. and M. J. Maher: 1994, 'Constraint logic programming: a survey'. *Journal of Logic Programming* **19/20**, 503–581.
18. Meier, M.: 1995, 'Debugging constraint programs'. In: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (*CP *'95)*. Cassis, France, pp. 204–221.
19. OADymPPaC, 'Tools for Dynamic Analysis and Debugging of Constraint Programs'. A French RNTL project. http://contraintes.inria.fr/OADymPPaC.
20. Schneiderman, B.: 1992, 'Tree visualization with treemaps: 2D space filling approach'. *ACM Transactions on Graphics TOG'92* **11**(1).
21. Schulte, C.: 1997, 'Oz Explorer: A Visual Constraint Programming Tool'. In: *Proceedings of the Fourteenth International Conference on Logic Programming (*ICLP *'97)*. Leuven, pp. 286–300.
22. Simonis, H. and A. Aggoun: 2000, 'Search-Tree Visualisation'. in [9], Chapt. 7.
23. Simonis, H., A. Aggoun, N. Beldiceanu, and E. Bourreau: 2000a, 'Complex Constraint Abstraction: Global Constraint Visualisation'. in [9], Chapt. 12.
24. Simonis, H., T. Cornelissen, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco: 2000b, 'Using Constraint Visualisation Tool'. in [9], Chapt. 13.
25. Smedbäck, G., M. Carro, and M. V. Hermenegildo: 1999, 'Interfacing Prolog and VRML and its Application to Constraint Visualization'. In: *The Practical Application of Constraint Technologies and Lo gic programming*. pp. 453–471.
26. Swedish Institute of Computer Science: 1991–2003, 'SICStus Prolog v3 User's Manual'. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden. http://www.sics.se/isl/sicstus.html.
27. Van Hentenryck, P.: 1989, *Constraint satisfaction in Logic Programming*. MIT Press.