facultad de informática

universidad politécnica de madrid

VIFID User's Manual

J. M. Ramos   Manuel Carro

TR Number CLIP3/98.0

VIFID User's Manual

Technical Report Number: CLIP3/98.0

Authors

J. M. Ramos
M. Carro
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid — Spain

Keywords

Constraint Logic Programming; Prolog; Program Visualization.

Acknowledgements

Thanks to all the members of the CLIP Group and the rest of the partners of the DISCIPL Project for useful discussions and feedback on the tool.
This work has been partially supported by the European ESPRIT LTR project DISCIPL # 22532 and Spanish CICYT Projects E96-1015-265 and TIC96-1012-C02-01.

Abstract

This is the user manual of VIFID, a tool to visualize the data evolution of CLP(FD) programs. VIFID is instrumented as a Prolog library which uses Tcl/Tk as graphic interface, and depicts in an intuitive way the state of the constraint store at selected points in the program. It has been developed and tested under the X Window environment. It gives the user several facilities, including allowing to post/unpost arbitrary constraints during the execution of the program. Little penalty is imposed regarding execution speed when visualization is turned off.

Resumen

Este es el manual de usuario de VIFID, una herramienta destinada a la visualización de la evolución de datos en programas CLP(FD). VIFID está realizada como una librería de prolog que usa Tcl/Tk como interfaz gráfico, y muestra de un modo intuitivo el estado del almacén de restricciones en puntos seleccionados de un program. Se ha desarrollado y probado bajo en el entorno de ventanas X Windows. Ofrece diversas posibilidades al usuario, incluyendo el añadir y eliminar restricciones arbitrarias durante la ejecución de un programa. La velocidad de la ejecución de los programas que usen esta libraria apenas cambia cuando no se está dibujando activamente.

Contents

1 Introduction

VIFID is a tool intended to prove different visualizations of constraint logic programs, aiming at being used both for teaching and debugging.

The tool is implemented as a Prolog library which provides spypoints, which are placed in the source program, so that each time a spypoint is reached, a set of windows shows information about the constraint store through the depiction of selected variables, and control is transfered to a top-level window from where the execution can be followed automatically or step by step.

Each value in the domain of a variable is represented as a square. Active values are shown in green and the others in red (respectively, clear and dark grey in a black and white picture). A line of squares represents the whole domain of a variable (Figure 1).

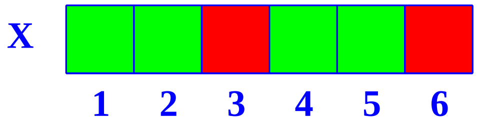

Figure 1: Domain of a variable

User constraints can be added at runtime either by clicking with the mouse on an active value of a variable, or by hitting the "Input user constraint" button and typing in the constraint in the window which pops up.

The current implementation supports only finite domains. It is planned for the future to be integrated with APT [Lue97].

This manual is organized as follows: Section 2 describes the main environment; Section 3 shows the different visualizations currently implemented. Section 4 describes how to post and unpost constraints. Section 5 contains an example in which the capabilities of VIFID are shown. Finally, Section 6 is a list of the software needed to use VIFID.

2 Starting a session

VIFID is available as a library module which is accessed by using the directive:

```
:-use_module(traceclpfd)
```

There are some predicates (`init_state/3`, `show_state/2`, `show_state/1`, `final_state/1`) which must be added to the source program for visualizing finite domain constraints. See examples in Section 5.

init_state/3 creates the main window and initializes the internal data structures with the variables used in the constraints and their initial domains. It also initializes all the global variables used during execution. It has 3 arguments: the constrained variables, the names of those variables, and a free variable where an internal structure is returned.

show_state/[1,2] acts as breakpoint which transfer control to the code of the visualizer. `show_state/2` calls internally to `show_state/1`, but it also adds a label to that breakpoint.

final_state/1 tells the visualizer code that the execution has finished.

Upon reaching `init_state/3` the main window of VIFID appears; this window is divided in two parts: the button area and the visualization area.

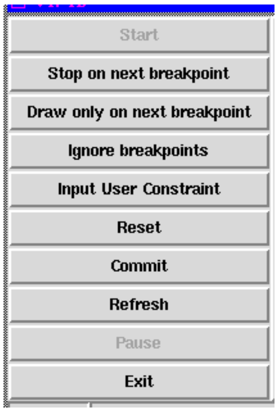

Figure 2: Button area

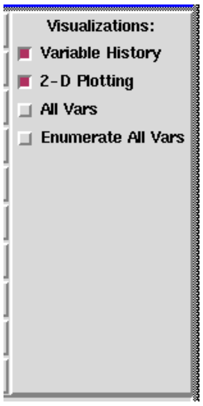

Figure 3: Visualization area

The button area, in Figure 2, allows triggering the different actions that can be performed during program execution. Buttons are enabled/disabled depending on the execution point, in order to protect the program from inconsistent states. In the visualization area (see Figure 3) the user can select some (or all) of the visualizations implemented. Clicking on a visualization at the visualization area of the main window has the same effect as clicking on the "Close" button when that visualization is active. If it is not active when selecting it, a new window pops up, and the next breakpoint will show domain values of the variables in it. We will describe all the visualizations and their meaning in the next section.

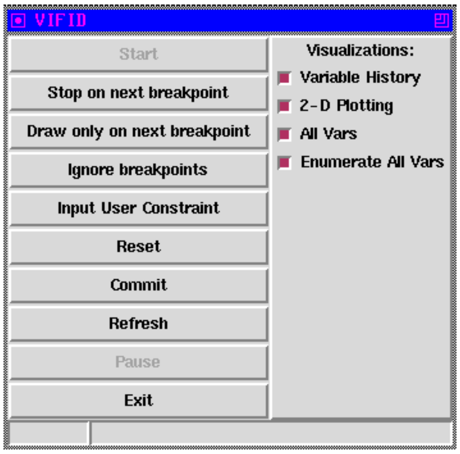

Figure 4: Main window

Pressing the "Start" button, after selecting some visualizations[1], initializes the environment. Then the visualization windows pop up and they show the variables and their initial domains. The execution is then stopped until a button is pressed.

Now we must select the execution mode. There are three buttons which select the execution mode of the program:

- "Stop on next breakpoint": each time a `show_state/1` or `show_state/2` predicate is called, the current domain of the variables is drawn in the active windows and the program execution waits for a user action. This is the initial execution mode, called **Waiting**.
- "Draw only on next breakpoint": when the predicates `show_state/1` or `show_state/2` are called, the current domain of the variables is drawn in the active windows, and the program continues until the end of the execution, or until the "Pause" button is pressed. This execution mode is called **Drawing**.
- "Ignore breakpoints": the predicates `show_state/1` and `show_state/2` are ignored. The current domain of the variables is not redrawn, and the program goes on until the end of the execution or until the "Pause" button is pressed. This execution mode is called **Skipping**.

The current execution mode is shown in the main window at downleft. In the **Waiting** mode the following buttons (and associated actions) are active:

- "Input user constraint": a window appears and the user can type a constraint in it (see Section 4 for further details). When the constraint is posted the current domains of the variables are drawn in the active windows.
- "Reset": constraints externally added by the user, and which were not commited to, are unposted, and the current domain of the variables is redrawn.
- "Commit": constraints externally added by the user are validated and cannot be eliminated. After this button is pressed, the "Reset" button has no effect.
- "Refresh": the current domain of the variables is redrawn in all the active windows.

If the execution mode is **Drawing** or **Skipping** the user can only press the "Pause" button. When that is done the execution mode changes to **Waiting** and the tool listens again to the rest of the buttons in the panel.

There is also an "Exit" button which closes all the windows and stops the execution at the current state if it has not finished yet. At the bottom of this main window there are two state labels:

- Execution mode: shows the current execution mode (**Waiting**, **Drawing** or **Skipping**).
- Status: shows the label associated to the breakpoint[2] more recently executed. Error messages that can appear while constraints are being posted are also shown here.

[1] The visualizations used during program execution can be selected/deselected at any moment at runtime.
[2] Only if the `show_state/2` predicate is used.

3 Visualizations implemented

There are four visualizations in the current implementation of the tool. This section will explain what is shown in each of them and how user interaction is performed.

As explained in Section 1, each value in the domain of a variable is represented by a square. A line of squares represents all the possible values of a variable. A green square means that value is still active, i.e, a value not yet removed from the variable domain; this value is currently a possible solution. A red square means that that value does not satisfy the current constraint store (see Figure 1), and therefore it has been removed from the variable domain.

The leftmost square represents the minimum value of the domain of a variable, while the rightmost square represents the maximum one. The rest of the values are sorted in ascending order.

All the visualizations have a "Close" button that destroys the window when clicked on.

At startup, the windows resize automatically to accomodate the visualization. However, the maximum initial size of each window is limited to 800×600 pixels.

3.1 "Variable History" visualization

This visualization represents the history of the selected variables as the program executes, keeping a time-oriented track (Figure 5).

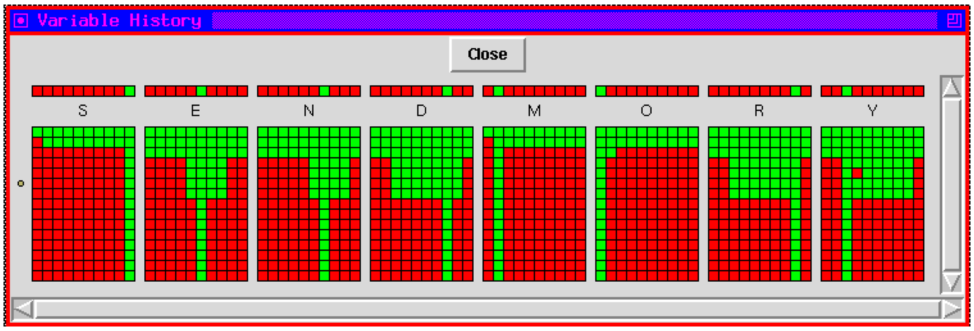

Figure 5: Variable history visualization

The line of squares that appears on top of the displayed area represents the current domain of the variables shown, and the name of the variables (given in `init_state/3`) is shown below this line. The rest of the lines show the domains of the variables at each breakpoint call. Time is shown vertically, running from top to bottom.

Some icons may appear on the left of the variable history depiction, namely:

- a hooked arrow, meaning backtracking. A failure has happened due to an inconsistent state, and a previous state is recovered and execution restarts there.

- a brown dot, meaning that this line was explicity redrawn. This is done in order to mark lines which do not follow the normal sequence of execution, but rather that those lines have been drawn in order to make all visualizations consistent.

When the program finishes the bottommost line shows the final domain values (and the top line too). At the end of a successful computation, all domains have (usually) become singletons.

There are also scroll bars to navigate through the displayed area: this is needed if the number of variables and their domains displayed are bigger than the size of the window.

3.2 "All Vars" visualization

This visualization represents the current domain values of the variables inspected at each breakpoint.

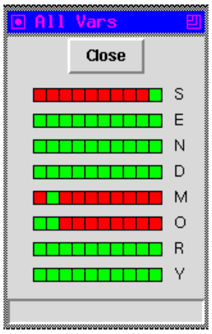

Figure 6: "All variables" visualization

The current domain rendered is the solver domain. This means that it has been obtained using CLP language primitives, and it reflects the internal idea of the solver about the current domain of the variables, which is not necessarily the most constrained one.

Each line represents a variable with its name at the right. A status line appears at the bottom of the window. Two messages can be written to this area:

- `constraint(variable,value)`: it is shown as the result of posting a user constraint when the user constraint is placed by clicking on a square of the depiction. The value removed from the domain of the variable is displayed.
- `Failed constraint`: a constraint which would have made the constraint store to be inconsistent has been posted.

3.3 "Enumerate All Vars" visualization

This visualization is similar to the "All Vars" visualization, which was described before. The difference between these two visualizations is how the current domain of the variables is obtained. In this case, the domains are accessed by enumeration: each possible value of the variables is tested for consistency, and only those values consistent with the store are reported as such. This visualization has been implemented in order to show the solver completeness (as it can be observed when comparing Figure 6 with the visualization in Figure 7 which corresponds to the same state of a program). "All Vars" shows the internal domain values of the solver, while "Enumerate All Vars" calculates which of these domain values can be a solution in the current constraint store because they satisfy all constraints.

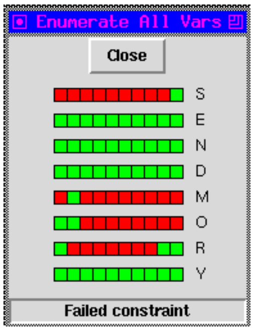

Figure 7: "Enumerate all variables" visualization

3.4 "2-D Plot" visualization

This visualization shows the relationship between two variables.

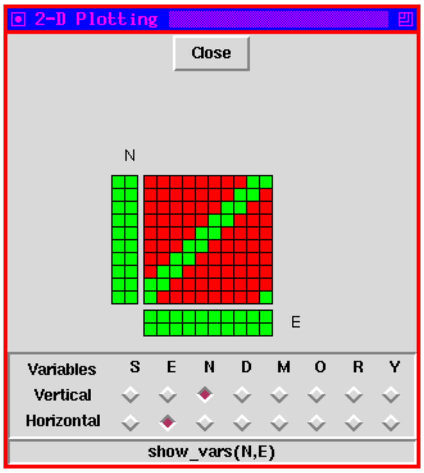

Figure 8: 2D Plot visualization

The window for this visualization (Figure 8) can be divided into three parts:

- Selection area: the variables to be depicted in the 2-D visualization can be chosen here (see Figure 9). There are three labeled lines. The first line shows the name of the variables we can inspect (those which were initially selected). The second line allows selecting the variable to be displayed vertically; and the last one is used to select the variable which will be displayed horizontally. Each time a variable is selected to be represented, the grid is modified to reflect the current values.
- Grid area: This is the main part of the window. Here, the current domain values of two variables are displayed. One of the domains is printed vertically and the other one horizontally, with the names of the variables beside them.

  The leftmost line (vertically) and the bottommost (horizontally) represent the current domain of the variables as kept by the solver. These values depend on solver completeness, and are obtained by accessing CLP language primitives, as in the representation

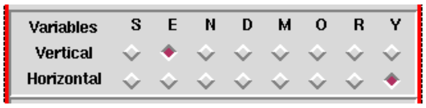

Figure 9: Selection area

"All Vars" of section 3.2. The lines placed by them, proceeding inwards, represent the current domain values of the variables. These are obtained by enumeration; so there must be the same or less active values in this lines than in the previous ones. In those lines the values are sorted bottom to top and left to right.

A grid made up of squares is enclosed by the two axes, and represents the relationship between the variables selected. The grid is worked out by enumeration.

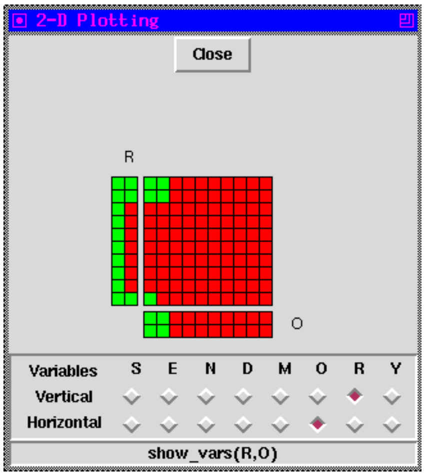

Figure 10: Relationship between two selected variables

In this area it is also possible to post constraints (see Section 4 for further details).

- Status line: the last action performed in the window is displayed here. Three different messages can appear:
  - ``show_vars(A,B)'' means that A and B are the last variables selected to be drawn in the grid area. The domain values of these variables and the relationship between them are displayed. Variable A is shown vertically and variable B horizontally.
  - ``constraint(var,value)''. A new constraint has been posted by clicking on the square that represents an active value of a variable. This message tells us that value has been removed from the domain of the variable.
  - ``Failed constraint''. A constraint inconsistent with the current store has been posted. The constraint store is not affected.

4 Posting and unposting constraints

Constraints other than those present in the program can be added (and removed afterwards) from the constraint store. There are two different ways of doing that:

- When the "Input user constraint" button is clicked on, a new window appears, as in Figure 11. In this window, constraints can be typed in, using the same syntax as in the source program, and with the names of the variables as appear in the visualization. When the "Return" key is pressed, or the "OK" button is clicked on, the constraint is added to the store, and all the active visualizations are updated.

  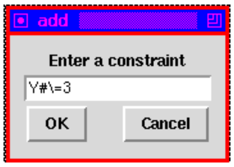

  Figure 11: Add a user constraint window

  If an invalid constraint[3] has been typed in, an error message appears in the main window. If that constraint causes an inconsistent state in the constraint store an error message appears in the VIFID main window again.

  The other windows remain blowed when the "Add a constraint" window is active.

- Inequality constraints can be more easily posted just by clicking on any of the squares that represent the domain of a variable. When one of these squares is clicked on, an inequality constraint (variable 'X' cannot be equal to value 'V'[4]) is posted. This can be done currently when visualizing "2-D Plot", "All Vars" and "Enumerate All Vars". When a square with an active value is clicked on, its color turns to black to show the value constrained. Then, if this constraint affects other variables, the values removed from the active domain of the affected variables will be shown in yellow (see Figure 4). This helps in depicting the relationship between variables. Of course, the constraint store must be consistent. Otherwise, a message appears in the status line, and the constraint is not added to the store.

These constraints posted at runtime may be meant to check a state, in which case we do not want them to be effective any longer. In that case, the "Reset" button can be pressed to eliminate the newly added constraints. If you go on executing instead, these constraints will not be removed when the "Reset" button was pressed.

5 Example

This is an example of how a program can be instrumented for its use with VIFID. The program we are going to use is in Figure 13.

[3] We consider a constraint invalid when it has a syntax error, any variable does not exist, ...

[4] 'V' is an active value of the variable, represented by a square, where the user clicked on.

Figure 12: Posting a user constraint by clicking on the domains of the variables

This program calculates the location of N queens on a N×N board so that they do no attack each other.

The main predicate is `queens(+N, -Qs)`. `N` is an integer with the size of the board and `Qs` is a list with the location of all the queens.

The usual souce code has been enriched with directives to load the tracing library (`use_module(traceclpfd)` in the second line), and with calls to primitives aiming at showing the state of the variables.

The calls `init_state/3`, `show_state/2`, `show_state/1`, `final_state/1`, marked with a '⇒', initialize the environment properly and show the domains of the variables in the active windows at the points we are interested in showing them.

The predicate `constrain_values/4` initializes the domains of the variables and posts the needed constraints, visualizing the constraint store every time a queen is constrained against the rest.

The predicate `my_labeling/3` is used instead of `labeling/2` to obtain the solution of the problem. A call to `show_state` is issued after every step in the enumeration, to show the domain of the variables while the solution is being generated.

After loading the code we execute the query:

```
?- queens(8,Qs).
```

Then a window as in Figure 14 appears. We can select here the visualization(s) we want to use during the execution and then press the "Start" button. The main window will become as in Figure 4, and the visualizations selected appear as in Figures 15, 16, 17 and 18.

By clicking on the "Stop on next breakpoint" button, we can see different snapshots of the execution. Figures 19 and 21 show the relationship between two variables in two different execution moments. Figure 20 (which resembles very faithfully the chessboard in which queens are being placed) shows the domains of the variables from the point of view of the constraint solver. Finally, Figures 22 and 23 highlight the differences between visualizing a solver-maintained domain and a enumeration-obtained domain. It can be observed that the "Enumerate All Vars" visualization shows less domain values of the variables than those in "All Vars" visualization, due to the different methods used to get the domain values (see sections 3.2 and 3.3); this difference can be used to acquire an intuitive idea of the strength and accuracy of the internal FD solver.

```
  :- use_module(library(clpfd)).
⇒:- use_module(traceclpfd).
  queens(N, Qs):-
      queens(N, Qs, []).
  queens(N, Qs, Type):-
⇒     list_of_vars_names(N, Qs, Names, 1, N),
⇒     init_state(Qs, Names, St),
⇒     show_state(St, 'initial'),
      constrain_values(N, N, Qs, St),
⇒     show_state(St, 'constrained'),
      all_different(Qs),
⇒     show_state(St, 'All different'),
      my_labeling(St, Qs, Type),
⇒     final_state(St).
  my_labeling(St,[Q|Qs],T):-
      labeling(T,[Q]),
⇒     show_state(St, labeling),
      my_labeling(St,Qs,T).
  my_labeling(_St,[], _T).
  constrain_values(0, _N, [], _St).
  constrain_values(N, Range, [X|Xs], St):-
      N > 0,
      X in 1 .. Range,
      N1 is N - 1,
      constrain_values(N1, Range, Xs, St),
⇒     show_state(St, 'step in constraining'),
      no_attack(Xs, X, 1).
  no_attack([], _Queen, _Nb).
  no_attack([Y|Ys], Queen, Nb):-
      Queen #= Y + Nb,
      Queen #= Y - Nb,
      Nb1 is Nb + 1,
      no_attack(Ys, Queen, Nb1).
```

Figure 13: N-queens source program

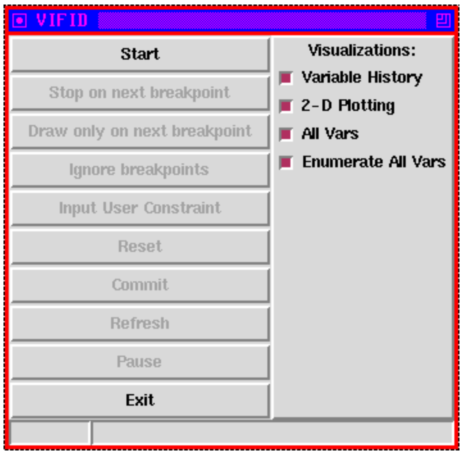

Figure 14: The main window

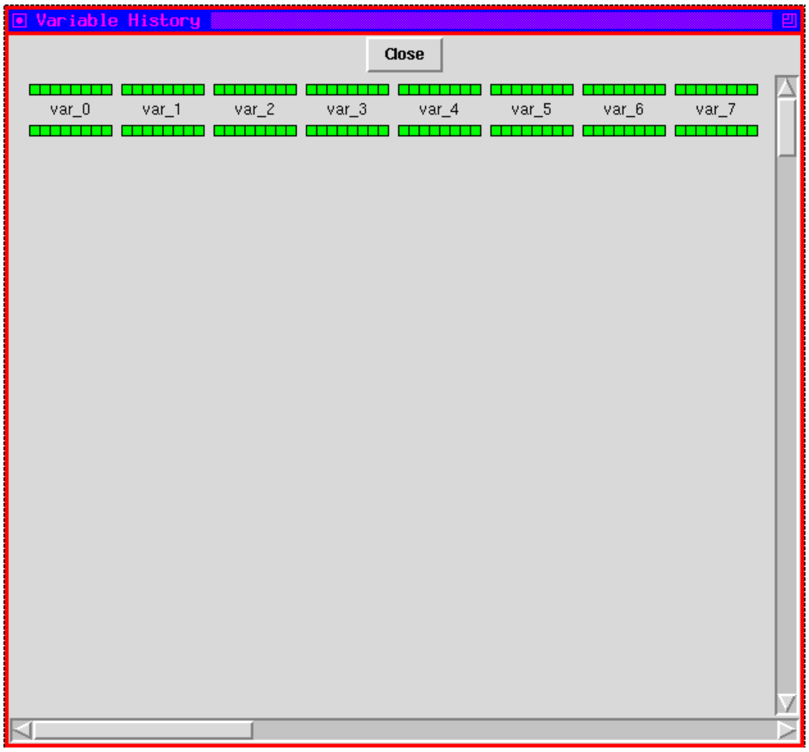

Figure 15: Variable History visualization at the beginning

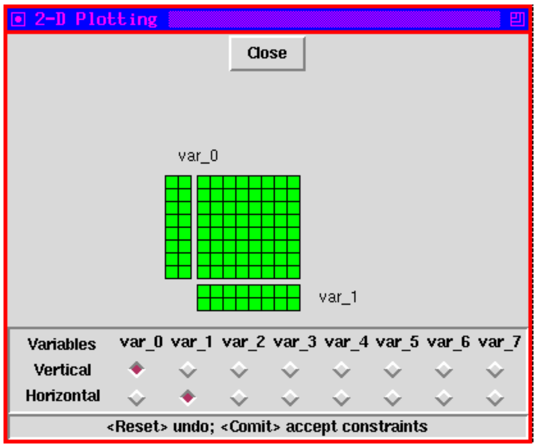

Figure 16: "2-D Plotting" visualization at the beginning

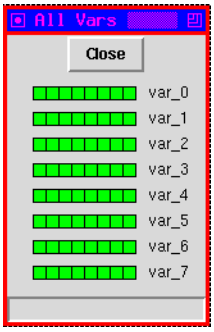

Figure 17: "All Vars" visualization at the beginning

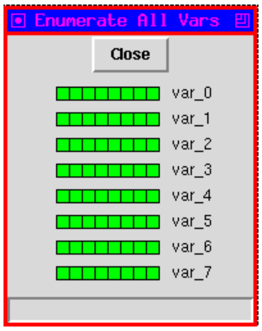

Figure 18: "Enumerate All Vars" visualization at the beginning

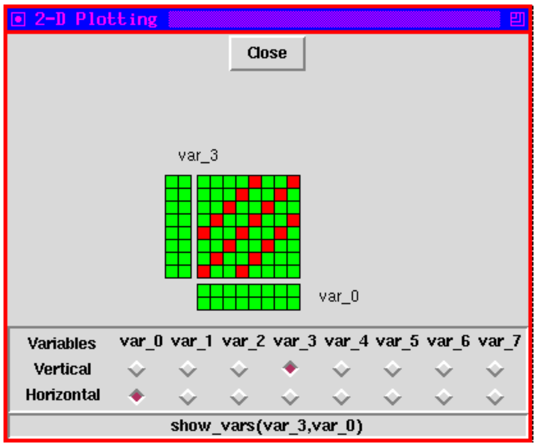

Figure 19: 2-D plotting visualization while adding program constraints

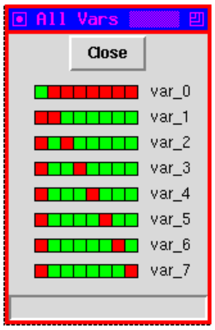

Figure 20: All vars visualization before labeling

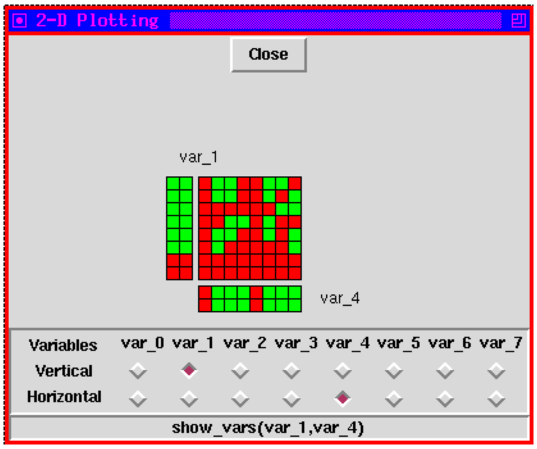

Figure 21: 2-D plotting visualization before labeling

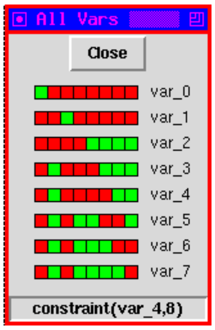

Figure 22: "All Vars" visualization while labeling

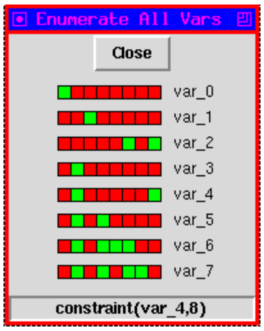

Figure 23: "Enumerate All Vars" visualization while labeling

As explained before, there are two ways to post constraints: directly, by clicking on a value of the domain of a variable, and by using the "Input user constraint" button (see section 4). Figures 24, 25, and 26 show the result of posting constraints directly. In the visualizations "All Vars" and "Enumerate All Vars" the same constraint has been posted, but in "All Vars" no value has been affected by the constraint, while in "Enumerate All Vars" some values appears with a lighter colour. Figure 27 shows the window where the user can input constraints.

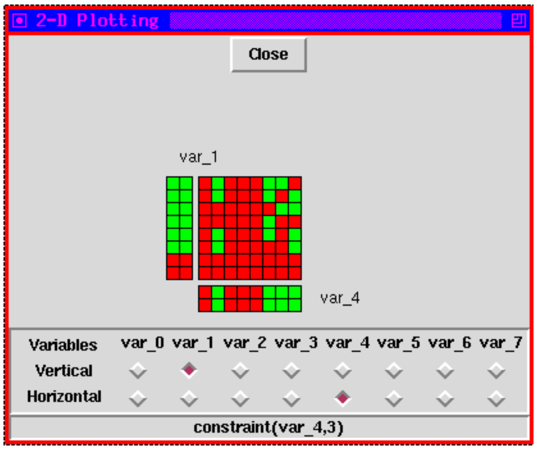

Figure 24: Posting a user constraint directly in 2-D plotting visualization

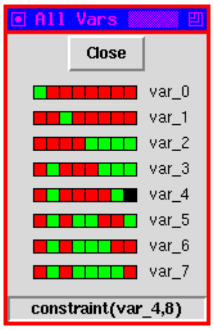

Figure 25: Posting a constraint in "All Vars" visualization

Figure 26: Posting the same constraint in "Enumerate All Vars" visualization

Figure 29 shows empty domains for some variables. That means that a failure will follow because the constraint store will become inconsistent. Figure 30 shows the same situation. There is no value of the two selected variables which can be a solution. The result of the *backtracking* which inmediately follows is shown in the piece of the "Variable History" visualization (Figure 31).

At the end of the execution all the active visualizations show the final solution. At this point, the user can further constraint the problem if any of the variables has more than one possible value, in order to get a definite solution. Figure 32 shows the history of the program

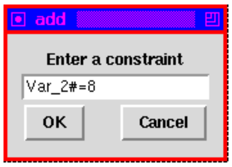

Figure 27: Posting a input user constraint

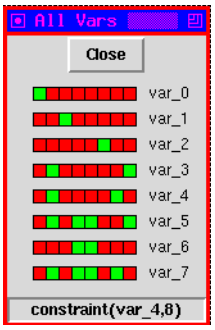

Figure 28: "All Vars" visualization before backtracking

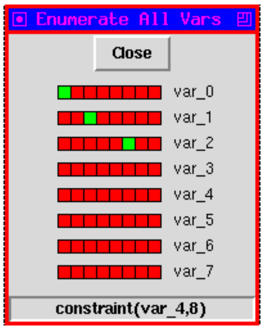

Figure 29: "Enumerate All Vars" visualization before backtracking

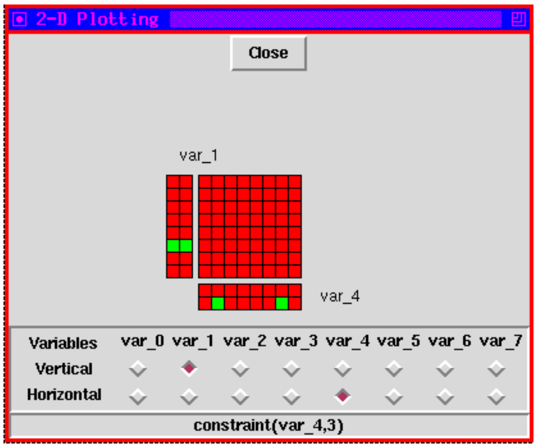

Figure 30: 2-D plotting visualization before backtracking

Figure 31: Variable History visualization with backtracking

execution. On the left of the window we can see when a failure has happened, and the domain values of the variables change to prove another solution.

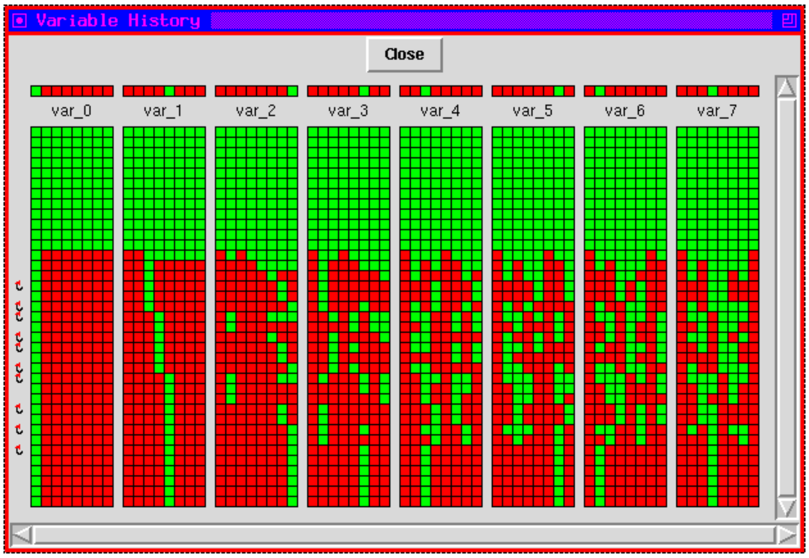

Figure 32: Variable History at the end

The user can, of course, see the text-based solution as traditionally given by CLP systems:

```
| ?- queens(8,Qs).
Qs = [1,5,8,6,3,7,2,4] ?
yes
```

6 Software required by VIFID

VIFID has been developed using standard Prolog Systems, so that it can be used at any site without worrying about portability. The main elements neccesary are:

- Prolog (not special features are needed; it only must support mutables). SICStus Prolog version 3.6 has been used in this implementation.
- CLP(FD) capabilities. Previous versions of SICStus Prolog did not have certain CLP(FD) primitives (as `fd_set/2`).
- TCL/TK interface. No special TCL/TK extensions are needed. TCL 7.4 and TK 3.6 were used in this implementation.

Bibliography

[Lue97] A. López Luengo. Apt: implementación de un visualizador gráfico de la ejecución de programas lógicos. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, October 1997. In Spanish.