1 VISUAL TOOLS FOR DEBUGGING PROLOG IV PROGRAMS

(version 1.0b - September 1998)

IN THIS CHAPTER will be found a description of the Prolog IV Visual Debugger and how to use it through examples.

1.1 CONTEXT

Prolog IV development system has a debugger to follow details of the execution of user programs. The debugger is based upon the so called Prolog IV Box Model Debugger which belongs to the Prolog IV core and dispatches information for various viewers.

Prior to this version, the user-visible part of the debugger was made of the two following execution viewers, running synchronously:

- the box model viewer,
- the source level viewer.

This new Prolog IV version incorporates a new viewer, working synchronously with the other viewers.

It is to be notice that it is not mandatory to run all viewers to make them work. The ones that are unneeded at a given time can be shut off to avoid too much information to be given to the user and/or loosing CPU time in displaying them. (A prolog-like engine, even in debug mode is much faster than I/O he produces.)

1.2 VISUAL DEBUGGING

The main tool to visualise graphically a running Prolog IV program is the Execution Tree viewer.

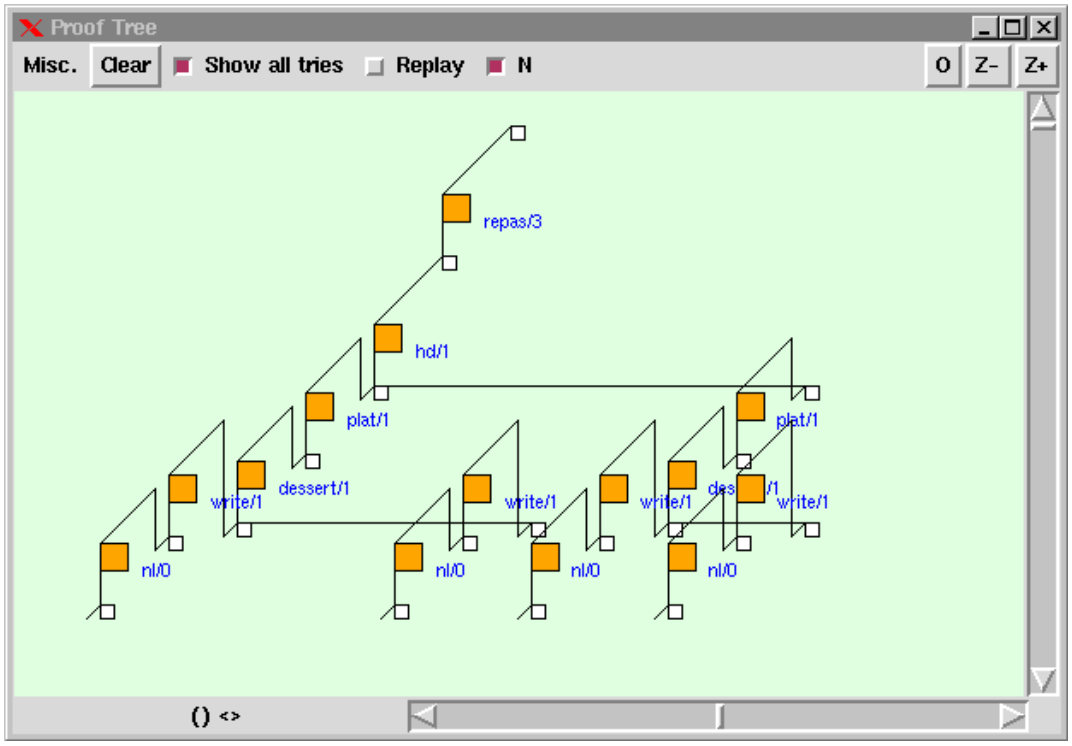
Its purposes can be summarised as follow:

- to give a full view of an execution tree
- to give a view of a proof
- to go quickly to a given place of the execution tree
- and help restarting the execution until a given point is reached.

The Prolog IV Visual Debugger is partially founded by ESPRIT Project « DiSCiPl » number 22532.

1.2.1 Giving a view of an execution tree

Here is a typical (although small) debugging session with the Execution Tree viewer:



This main canvas displays a short execution tree of a program.

This is an AND/OR tree, with a graphical information about the level of each AND node.

This tree is intended to be seen as a three dimensional tree display, in perspective. We are “above” the general plan of the tree.

Various boxes appears with predicates names and connections between them.

1.2.2 What are all these small boxes ?

The reader is supposed to know enough of the Prolog IV debugger, or at least, some of the basics upon the box model debugger.

A big coloured box is a call-box (once per set of rules). For now, only call-boxes can have a name aside (to avoid too much text in the canvas).

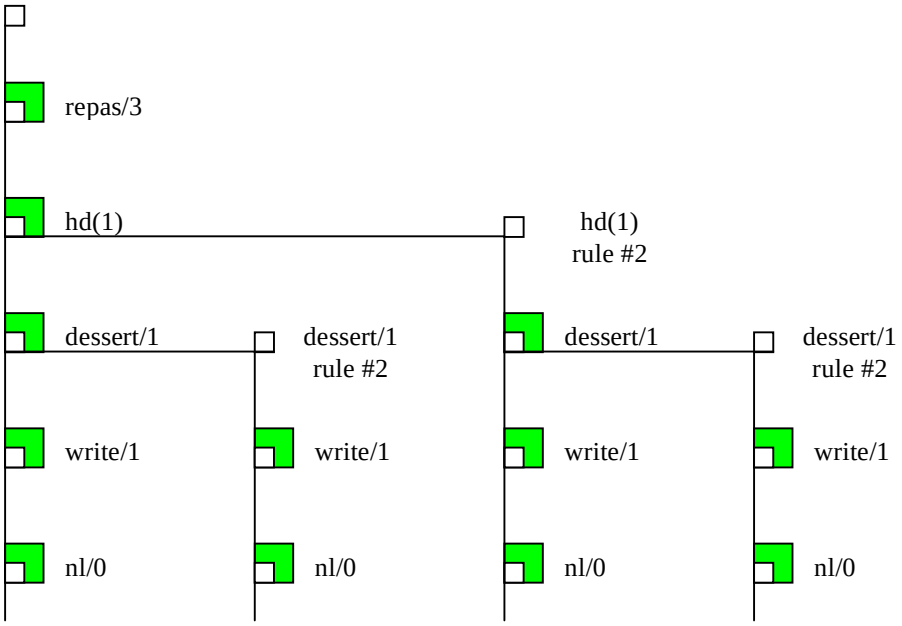
A small white box is a rule-box and shows an attempt to match the call to the head of a rule. Only successful matches are displayed if the button “Show all tries” is checked.

1.2.3 How to understand them ?

Here is a view “from above” (a view not provided in the debugger) of the previously seen execution tree. It is also supposed that boxes are flattened.

Alternatives (OR nodes) are going from left to right.

AND nodes are going from top to bottom.



In order to give more explanation about the view, we need to introduce the small prolog program whose execution is displayed:

```
to_begin :- repas(H,P,D), write([H,P,D]), nl.

repas(H,P,D) :-
   hd(H),
   plat(P),
   dessert(D).

hd(pate).
hd(salade).

plat(viande).

dessert(glace).
dessert(fruit).
```

With the above explanations and conventions, it is clear how executing goal “`to_begin`” could lead to the given execution trees.

As we know that successful calls go down, we don't know how to distinguish calls: in the “flat” tree display, one can not guess that calls

`write/1` and `nl/0` belong to the same rule's body and that `write/1` and `dessert/1` don't.

This is the purpose of the “non flat” tree view to give some indication about the level of every call.

1.3 WINDOW AND DESCRIPTION

1.3.1 Window controls description

1.3.1.1 Menu “Misc.”

It contains the following items:

- Load Exec. Tree ...
- Show Names
- Replay mode
- Replay-left mode
- Names >

1.3.1.1.1 Item “Load Exec. Tree ...”

This item asks the user to choose a file containing the description of a previously executed and saved tree and loads it in the canvas.

1.3.1.1.2 Item “Show Names”

This item set the canvas to display predicates names for every call boxes. This is useful only if the current zoom factor is at this time set to a reasonable value. Otherwise names are no more displayed; (since the font of the text is not scaled when zooming occurs, this prevents the names to be too much close to each other. Scaling down the font size, on the other way, would not be useful.)

1.3.1.1.3 Item “Replay mode”

This sets the Execution Tree Viewer in a mode where:

- the contents of the canvas will not be destroyed when the next query is launched.
- the tree can be used to choose both a temporary break-point and its proof branch,
- to allow the user to execute a goal which will runs on the previously made tracks until the break-point is reached.

Note: No more box drawing occurs during the execution.

1.3.1.1.4 Item “Replay-left mode”

If checked, (and if the user is in replay mode,) the execution of the next query will parse the whole tree at left side of the chosen branch.

Else, only this branch need to be re-executed. This could be done mainly if the program is pure-prolog i.e. there is no predicate with side effect (like I/O, asserts or global assignment) in the “left part” of the branch. Otherwise, let this option unchecked.

It may be important to know about which parts of the program are “safe” since running a single branch is normally much more faster than doing the whole thing, consuming time in trying known dead ends, giving unwanted solutions (since we are debugging elsewhere).

1.3.1.1.5 Item “Names”

The last item is a cascade menu made of predicates names appearing during the execution. The user can give a colour to each of them, with the effect that a chip of the chosen colour is displayed around drawn boxes with such a predicate names, in the whole execution tree.

1.3.1.2 Button “Clear”

To clear the whole canvas and forgetting everything about the current execution. This is automatically called when a new goal is launched from the Prolog console, unless box “Replay mode” is checked.

1.3.1.3 Check box “Show all Tries”

Its purpose is to set a boolean flag whose meaning is to display every attempt to match a head of a rule.

If the boolean is set, non matching head for a given call are also displayed; else, only successful matching heads are displayed.

Unsetting this flag will result in a significantly smaller execution tree for the next user query to be launched.

1.3.1.4 Check box “Replay”

Short cut for the “Replay mode” menu item. (*See above*)

1.3.1.5 Check box “Names”

Short-cut for the “Show Names” menu item. (*See above*)

1.3.1.6 Buttons “O”, “Z-” ,“Z+” and scrollbars

These controls act on the display occurring in the canvas.

Button “O” (for Origin) manages to make the origin of the tree to be visible at the upper left corner of the canvas.

Scrollbars can be used to move around in the canvas to see hidden part of the execution tree.

Using the middle button on a three-button mouse achieves the same result.

Button “Z+” zooms (with a factor of two); it scales all objects and distances in the canvas. Things are bigger and details of the tree are

more understandable, but less of the execution tree is likely to be seen at once.

Button “Z-” zooms down (with a factor of two); it scales all objects and distances in the canvas. Graphical objects are smaller, but more of the execution tree can be seen.

Zooming is cumulative. This allows big zoom factors (as powers of two).

1.3.2 The Zoom Window

The Zoom Window is a feature allowing to have both a general view at the execution tree and keeping a detailed view of some part of this tree. In fact the Zoom Window uses no zoom factor (or use a zoom factor of one, with respect to the initial default) but the default size for boxes, which makes the local structure of the execution tree readable and understandable.

The Zoom Window can be called in clicking left mouse button while pressing CONTROL key, somewhere in execution tree the user wants to zoom in. This makes appear the zoom windows, or changes its contents. What is zoomed is visualised in the Execution Tree Window by the mean of a yellow square.

This window has its own control for displaying predicates names or not.

1.4 WORKING WITH THE DEBUGGER

First of all, it is reminded that only program compiled with debug option can be usefully handled by the debugger. (A program not compiled with this option is seen as a single call - a black-box).

The user is invited to have a look at chapter *Environment* from the Prolog IV documentation to get information about the existing Prolog IV Debugger and what is relative to its invocation.

It is mandatory to use Prolog IV within its graphical development system to get the ability to launch the Visual Debugger.

1.4.1 Connecting from the Prolog IV Debugger

Prior a query to be debugged is launched, you have to set Prolog IV in mode debug. Then after launching the query, you should get the debugger prompt (in either the Console window or the Debugger window).

At this prompt, type the following bold text to enable the Execution Tree Viewer:

```
(DBG PIV) graph 1
```

The main window “Execution Tree” should show up immediately.

1.4.2 Setting a temporary break-point

CTRL-SHIFT- left button mouse on a box set a temporary break-point. It also displays (in green) the branch issued from the top (the query goals) to this box. Either a call box or a rule box can be chosen.

1.4.3 Replaying the execution

After a temporary break-point having been set in the previous step, check mode “Replay” in the Execution Tree Window.

Check item (if appropriate - see notes for this command) “Replay-left mode” from menu “Misc.”.

Provided the query is the same as the one which draft the execution tree, just launching it.

Note: if the execution is not the same, because not the same query has been fired or because the debugger has not been called from the beginning of the previous query, extra display will appear on the canvas.

1.5 MISCELLANEOUS

1.5.1 Information in the canvas

When the mouse is moved over a box, some details about the box are displayed at the bottom left of the window.

These information are, in a compound form:

- the predicate indicator (as name/arity)
- the rule number (only if we are on a rule box)
- the box number (between parenthesis, as (11))
- the level of the call (between angle brackets, as <3>)

1.5.2 Some related textual debugger commands

Here are some new commands related in some way to the Visual Debugger. They can be typed at the prompt of the Box Model Viewer.

1.5.2.1 graph v

where **v** is 1 or 0, sets or unsets the Viewer connection to the Prolog IV debugger.

1.5.2.2 rule n

being on a box, on a RULE port, changes the current choice (rule number in its set) to be **n** an integer.

Note: for internal purpose, the last rule of a set should be noted with a minus sign (thus, as a negative number).

1.5.2.3 gfile name

opens or creates a file with name **name** which is to be containing the log of the execution, allowing later the execution tree to be loaded in the Visual Debugger (see item “Load Exec. Tree ...”).

⋄♦⋄♦⋄♦⋄♦⋄♦⋄