1 Introduction

In an ideal world, computer programs would run correctly as soon as they were written, and there would be no need for debugging. Unfortunately, the real world is far from ideal. Very few programs behave entirely correctly the first time they are run, and for a large program to exhibit consistently correct behaviour without any need for debugging is practically unheard of.

The sources of program errors are many. They range from the simplest typographical errors to the most obscure and subtle conceptual errors on the part of the program designer. Even faulty compilers can manifest themselves as apparent program bugs. In each of these cases, it is necessary to have some method for systematically tracking down the source of the problems in the errant program. It is for this purpose that debuggers exist.

The design of a debugging system is in large measure a psychological problem. It is the human, attempting to understand and fix the program at hand, who is the critical component in the whole debugging process. Certainly the debugger must correctly present all relevant information, but this is of little use if the human is so overwhelmed by irrelevant detail that critical information cannot be extracted from the executing program. The display of information by a debugger must then resolve the tension between two opposing forces: in one direction to display everything that is happening in great detail and in the other to present to the user just those few things which are relevant. In the remainder of this section we review the most popular style of debugger used for Prolog and introduce DEWLAP (DEbugging With Logical APplications) as a highly graphical Prolog debugger [Dewar 1985, 1986]. A brief review is then made of graphical and semigraphical debuggers for other languages. Section 2 describes how DEWLAP represents an executing Prolog program and its control flow. Section 3 extends this form of graphical display to data structures. Section 4 discusses some of the problems of implementing DEWLAP efficiently and section 5 concludes with a summary.

Prolog debugging

Traditionally, Prolog debugging has been done by means of textual traces and the textual display of variable values [Byrd 1980, Naish 1983, Pereira 1978, Pereira 1984, Spacek 1982]. The four-port model of Prolog execution as described by Byrd [Byrd 1980] forms the basis for many commonly available Prolog debuggers.

Each call in a Prolog program is viewed as having four ports: a *call* port which is visited once when the call is made; an *exit* port which is passed through once each time the call succeeds; a *redo* port which is visited each time a new solution is requested as a result of backtracking; and the fourth *fail* port which is visited once when no more solutions can be found. Sometimes a fifth *neck* port is added which is visited each time the unification of the call with the head of a clause succeeds. The output from this type of debugger consists of a printout of the name of the current call together with its bindings as well as some indication of the context of the call. For example, a line in Figure 1 shows (in order) a unique number associated with each call (this is used each time one of its ports is displayed); the depth of the call within the execution tree; the name of the port; the call itself together with its parameters; and optionally a prompt requesting input to control the debugging session.

As can be seen, this approach to a debugging display has a number of drawbacks:

- The user is often overwhelmed with irrelevant detail, especially if the calls include large or complex parameters. This can be alleviated by carefully selecting which calls are to be displayed, but this requires that the user have a good idea of where the bug lies before starting, and if a vital call has been omitted, the program must be rerun before the information in that call can be retrieved.
- All items appear with the same degree of prominence, regardless of their actual importance. It is often difficult to disentangle where the parameters to a call start and end. For example, it can be difficult to

```
   (2) 1 Call: strans([[a,b,_0],[_7,c,d],[e,_14,f]],_27,_28) ?
   (3) 2 Call: transpose([[a,b,_0],[_7,c,d],[e,_14,f]],[[_41,_28|_43]|_44]) ?
   (4) 3 Call: skim([[a,b,_0],[_7,c,d],[e,_14,f]],[_41,_28|_43],_32853) ?
   (5) 4 Call: skim([[_7,c,d],[e,_14,f]],[_28|_43],_55) ?
   (6) 5 Call: skim([[e,_14,f]],_43,_68) ?
   (7) 6 Call: skim([],_78,_79) ?
   (7) 6 Exit: skim([],[],[])
   (6) 5 Exit: skim([[e,_14,f]],[e],[[_14,f]])
   (5) 4 Exit: skim([[_7,c,d],[e,_14,f]],[_7,e],[[c,d],[_14,f]])
   (4) 3 Exit: skim([[a,b,_0],[_7,c,d],[e,_14,f]],[a,_7,e],[[b,_0],[c,d],[_14,f]
])
   (8) 3 Call: transpose([[b,_0],[c,d],[_14,f]],_44) ?
   (9) 4 Call: set([[b,_0],[c,d],[_14,f]],[]) ?
   (9) 4 Fail: set([[b,_0],[c,d],[_14,f]],[])
   (10) 5 Call: skim([[b,_0],[c,d],[_14,f]],_86,_32895) ?
   (11) 6 Call: skim([[c,d],[_14,f]],_93,_94) ?
   (12) 7 Call: skim([[_14,f]],_104,_105) ?
   (13) 8 Call: skim([],_115,_116) ?
   (13) 8 Exit: skim([],[],[])
   (12) 7 Exit: skim([[_14,f]],[_14],[[f]])
   (11) 6 Exit: skim([[c,d],[_14,f]],[c,_14],[[d],[f]])
   (10) 5 Exit: skim([[b,_0],[c,d],[_14,f]],[b,c,_14],[[_0],[d],[f]])
   (14) 5 Call: transpose([[_0],[d],[f]],_87) ?
   (15) 6 Call: set([[_0],[d],[f]],[]) ?
   (15) 6 Fail: set([[_0],[d],[f]],[])
   (16) 7 Call: skim([[_0],[d],[f]],_123,_32937) ?
   (17) 8 Call: skim([[d],[f]],_130,_131) ?
   (18) 9 Call: skim([[f]],_139,_140) ?
   (19) 10 Call: skim([],_148,_149) ?
   (19) 10 Exit: skim([],[],[])
   (18) 9 Exit: skim([[f]],[f],[[]])
   (17) 8 Exit: skim([[d],[f]],[d,f],[[],[]])
   (16) 7 Exit: skim([[_0],[d],[f]],[_0,d,f],[[],[],[]])
   (20) 7 Call: transpose([[],[],[]],_124) ?
   (21) 8 Call: set([[],[],[]],[]) ?
   (22) 9 Call: set([[],[]],[]) ?
   (23) 10 Call: set([[]],[]) ?
   (24) 11 Call: set([],[]) ?
   (24) 11 Exit: set([],[])
   (23) 10 Exit: set([[]],[])
   (22) 9 Exit: set([[],[]],[])
   (21) 8 Exit: set([[],[],[]],[])
   (20) 7 Exit: transpose([[],[],[]],[])
   (14) 5 Exit: transpose([[_0],[d],[f]],[[_0,d,f]])
   (8) 3 Exit: transpose([[b,_0],[c,d],[_14,f]],[[b,c,_14],[_0,d,f]])
   (3) 2 Exit: transpose([[a,b,_0],[_7,c,d],[e,_14,f]],[[a,_7,e],[b,c,_14],[_0,d
,f]])
   (2) 1 Exit: strans([[a,b,_0],[_7,c,d],[e,_14,f]],[[a,_7,e],[b,c,_14],[_0,d,f]
],_7)
```

Figure 1. A textual trace of a Prolog program on an 80-character display

tell from Figure 1 that the procedure "skim" has three parameters.

- The use of numbers such as _14 to refer to unique instances of variables is visually very confusing. As an exercise, the reader might like to see how many times _14 occurs in Figure 1.

The context of a call is provided by the call number, by immediately-preceding calls, and by the depth of execution. A dump of the (active) stack is also typically available. Unfortunately, viewed at this level, control is definitely non-linear. The success of a low-level call may cause the success of a number of higher level routines and very quickly leave the trace itself textually and logically very distant after only a few lines of output. Failure and backtracking can cause even more confusing transfers of control. Information is available in the form of the numbers identifying calls and the stack depth, to help in disentangling this flow. However, it is not easily usable in the heat of a debugging session. For example, consider two adjacent calls in a clause:

... call1(X,Y), call2(X,Y)...

A line saying that call2 has just exited (succeeded) may be removed a very long way from the corresponding line for call1 in the debugging trace. To find the line for call1, one must search back to the point where the 'call' port of 'call2' was entered, and then the success of 'call1' will be found on the preceding line. It is very likely that the user's attention will need to move quickly amongst such adjacent and probably logically coupled calls. The difficulty of doing this is inherent in the linear nature of any textual debugging system.

We solve these problems by a hierarchical graphical display of the current execution tree and of call parameters, as shown in Figure 2. Items which are more deeply nested in the tree or in data structures are drawn in progressively smaller sizes. This permits the entire execution state to be always completely displayed. Items at the leaves may be too small for details to be seen but they are always accessible for magnification and detailed consideration. Variables bound to each other can be discovered by requesting that all instances be highlighted.

The port model allows considerable control over the amount of detail printed by a debugger. Typically, each port for a given procedure can be optionally printed or not, and the prompt asking for input can optionally be omitted. For example, large parts of a computation can be omitted entirely or printed without delaying to ask the user for input to the debugger. However, it is not easy to specify that these options be set or unset for a particular instance of a call, nor to interrupt asynchronously in the middle of a long piece of display and change them. Also, if the program itself does any input and output, this becomes entangled and confused with the debugging activity.

We solve this by making use of multiple windows, with the debugging occurring in a window completely separate from the program. With the use of a mouse, individual nodes can be pointed out and options set for them, as well as generically for all instances of a particular functor.

An important part of any problem-solving activity is the abstraction the user has of the program. In particular, the data structures in a program may have an interpretation which is not obvious from their representation. For example, a list of lists might be interpreted by the programmer as a two-dimensional board. Understanding a program may well involve reasoning at this abstract level. To accommodate such high-level views graphically, DEWLAP allows users to write their own data-structure display routines in a high-level graphics language [Cleary 1984, Dewar 1985, Dewar 1986].

Graphical debugging in languages other than Prolog

Many graphical tracing and debugging systems exist for languages other than Prolog. Some of these concentrate on tracing the flow of control in programs, while others emphasize the display of data. Graphical techniques are used to varying degrees. The extent to which the user can tailor the display or obtain different

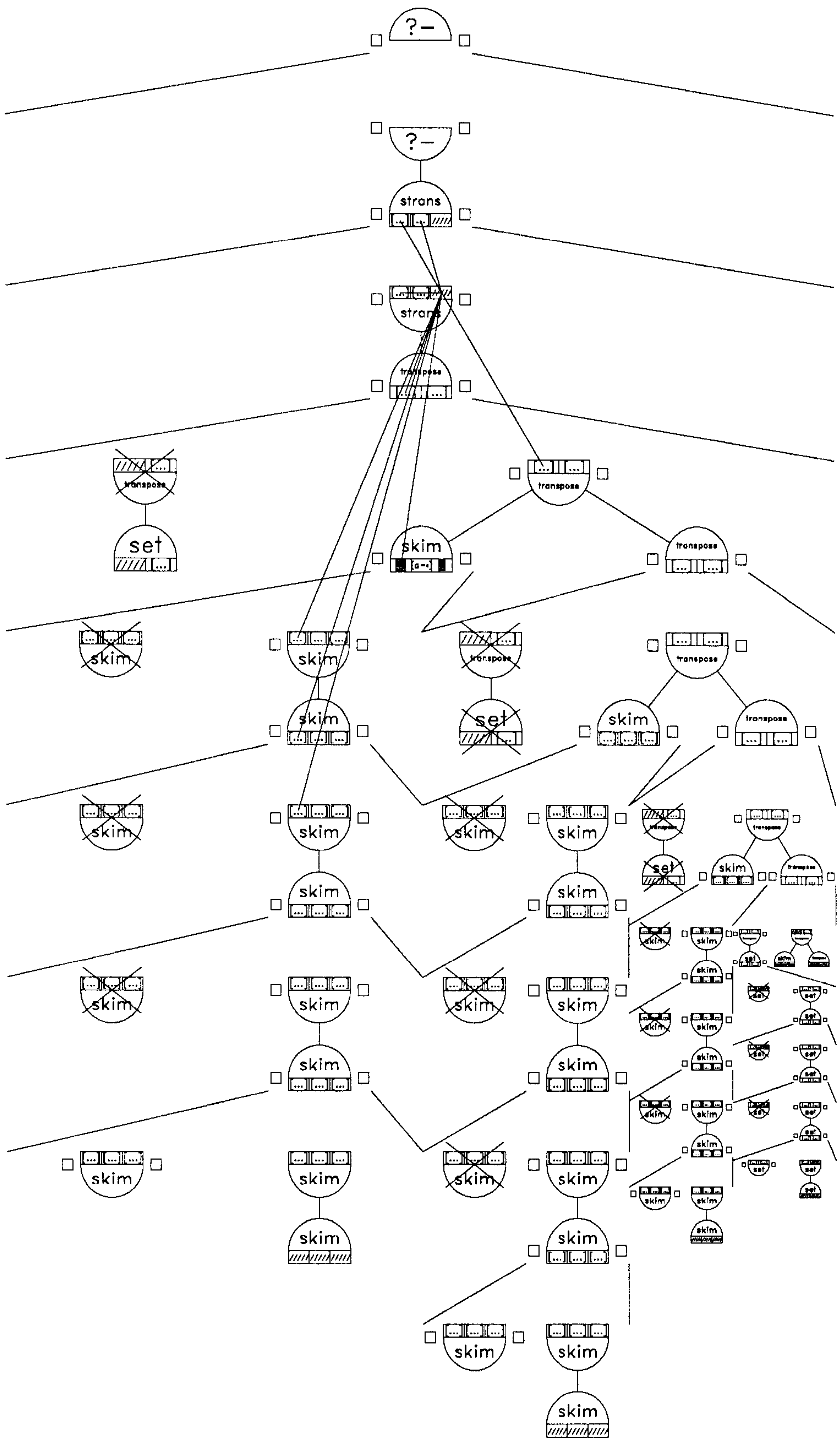


Figure 2. A DEWLAP display of Prolog execution in Figure 1

views of the same object also varies.

Much of what has been called "graphical" debugging has tended to be only semi-graphical in nature. Such debugging systems make use of terminals with cursor-positioning capabilities to display a small window on the source code, highlighting in some way (e.g., reverse video) the statement which is currently executing [Takahashi 1982, Weisling 1981, Teitelbaum 1981, Kline 1978, Shapiro 1974]. Such systems are a distinct improvement over purely textual traces, in that they allow the user to see not only which statement is currently executing, but also the context in which that statement occurs within the program. The view presented is also dynamic in nature, so that the user may concentrate on what is happening at the time, without having to keep track of where to look in a purely textual trace.

One of the shortcomings of this sort of semi-graphical trace is that the limited size of the user's display may make it impossible to show all the relevant parts of a control structure, such as a while loop. For example, a loop termination condition may not be displayed when the last few statements of the loop are being executed — a time when the user might strongly wish to see whether or not the loop will exit.

The Cornell Program Synthesizer [Teitelbaum 1981] circumvents this problem to some degree by basing its tracing on templates, rather than on individual statements. Each template corresponds to a conceptual program structure, such as a while loop, an if statement, an else clause, etc. By including comments with the templates and selecting an ellipsis mode, the user can have the actual source suppressed and only the comment displayed. This allows a higher-level view of the program to be taken, assuming the comments accurately reflect the actions of the elided source code.

PIGS [Pong 1983] is a system for Programming with Interactive Graphical Support. This system is built around Nassi-Shneiderman diagrams [Nassi 1983], with all programming, execution, editing, tracing, and debugging done within this context. As a program executes, the currently-executing block at any time is highlighted. This system allows the user to follow in detail the flow of control in a program, as long as control remains within one procedure. However, it is more difficult to follow what is happening if procedure invocation becomes very deep, as no provision is made for displaying the context within which the currently executing procedure was invoked.

One of the possible features of semi-graphical tracing/debugging that many such systems do not implement is the display of the values of variables. One which does implement such a facility is IVF, an Interactive Visual Fortran debugger [Shapiro 1974]. IVF shows the values of any variables used or changed in the statement currently being executed. This provides the user with complete information on the variables being directly used at any given point, but does not allow for the display of values of variables which may be of greater interest, such as active loop indices which do not appear in the current statement.

Schach has implemented a semi-graphical array trace package for displaying the values of portions of two-dimensional arrays in Fortran programs [Schach 1982]. The Fortran source is not displayed during execution, but the display does show the source line number of any statement which updates any of the arrays being displayed. The array display itself is updated whenever an assignment takes place to any element which occupies the upper-left or lower-right corner of the display of that array (i.e., when an array boundary element is changed); whenever an entire array is updated, as by a READ statement; or whenever thirty assignments to array elements have occurred, so that a significant portion of the array is likely to have changed.

The Cornell Program Synthesizer allows for the display of specified variables, with the display automatically updated when these variables are changed. If more variables are being tracked than there is room to display, the least-recently updated variable will be removed from the display to make room for a variable currently being updated, if that variable is not currently in the display.

If true graphical output is available, the problem of fitting appropriate amounts of program source code on the display is somewhat alleviated. Since text can be displayed in arbitrary sizes, more can fit in any given space. More important text or higher-level constructs can be shown in larger type, and less important details can be shown in smaller type or as bit patterns which can be zoomed in on to reveal the detail at the user's request.

The display of data structures can also be made much more intricate. Whereas semi-graphical display of data tends to be limited to displaying numbers as their decimal representation, pointers as decimal integers, enumerated types as names, and so on, graphical data display allows for arbitrarily complex representations. For instance, an array of numbers might be shown as a bar graph; a pointer could appear as a line with an arrowhead, connecting one part of the display to another part where the data pointed to is shown; an element of an enumerated type could be displayed as an icon which represents what such an element looks like in the real world; and aggregate structures could be represented by boxes with sub-regions.

One tracing package which has been implemented to display linked lists and graphs in an informative manner is GRAPHTRACE [Getz 1983]. This system deals only with Pascal records — other data types and program source tracing are not dealt with. When a program run under GRAPHTRACE encounters a breakpoint, the user may request that any allocated records be displayed. Records of different types are shown as different icons (square, diamond, etc.), and pointers of different types are represented by different kinds of lines (solid, dashed, etc.) with arrowheads. Thus, the display consists of a number of node icons interconnected by lines with arrowheads, representing the current state of the heap. It is possible to specify constraints on the direction in which pointers of a given type may be drawn (i.e., horizontally or vertically), so that some structure is imposed on the display. If the restrictions cannot all be met, the user is informed of this fact, since this failure could be the result of a program bug which caused an incorrect structure to be constructed.

The Computer Corporation of America has developed a Program Visualization System [Herot 1983, Brown 1985] which allows diagrams to be bound to C code. Selected variables are displayed whenever they are assigned to, in a format suitable for the variable type. Static diagrams and text may also be bound to the source, so that a user may, by pointing to a given portion of the source, retrieve graphical and textual documentation associated with it. This may be done at a number of levels of abstraction, potentially ranging from a low-resolution icon to a highly detailed diagram.

Two systems for displaying information about Simula executions have been developed at the University of Calgary. The first [Dewar 1984] allows Simula processes to be represented graphically. The currently-active process is highlighted, and processes move to appropriate places on the display whenever they enter or leave a queue. The user may request the display of detailed information about any displayed process or queue. The information which is displayed, and the manner in which this is done, may be tailored by the user.

The second Simula animation system is Andes [Birtwistle 1984, Joyce 1984]. This system presents a number of views of an execution, including the program logic view, the activity diagram view, the resource view, a high-level source trace, and a statistical display. The display formats are initially constructed by the user with the aid of a graphics editor.

A micro-PL/I system has been implemented by Ron Baecker [Baecker 1975] for use in the production of program-tracing demonstration films. This system could presumably be adapted for interactive use, given the appropriate display hardware. With the use of explicit pseudo-comment option selections, the user may cause this system to display specified data as it is updated, in any manner for which an appropriate display subroutine has been defined. Source tracing is not dealt with.

The LOOPS package [Stefik 1983] makes use of active variables to allow for the display of variables in a LISP-based system in a number of graphical ways. For instance, a number could be displayed as a dial, a meter, a vertical scale, a horizontal scale, or a combination of any of these with a digital readout as well. Whenever a displayed variable is altered, the display is automatically updated to reflect the new value.

Incense [Myers 1983] is a system for displaying data structures in Mesa. It does not allow for automatic updating of the data display as variables are changed, but must be explicitly invoked at a program breakpoint. Any specified variable may be displayed, in either a default or a user-specified format. Aggregate structures are shown as rectangles containing subfields for each component. Pointers are shown as smoothly-curving lines with arrowheads.

An interesting feature of Incense is that the display procedure invoked depends on the amount of screen space available for displaying the given data structure. Consequently, either an iconic representation or a detailed representation is used, as appropriate. An example of this in the default display format is the use of gray regions of appropriate length instead of text, whenever the text would not be readable in the space available.

The BALSA environment [Brown 1983, 1984] provides for the display of data in any number of user-definable formats, with defaults provided. Any given variable may be displayed in more than one window, with different formats for each. The displays are maintained in an object-oriented manner, with each view updating itself when the executing program informs it that an ''interesting'' event has occurred. The user may specify which program events are considered interesting. Zooming and panning facilities are provided. A similar system [London 1985] has also been implemented for Smalltalk.

2 Displaying the Prolog execution tree

As shown by van Emden and Kowalski [van Emden 1976], a Prolog execution corresponds to the construction of a proof tree. This suggests a very natural display format for tracing program executions, as shown in Figures 3 and 4. Following Ferguson [Ferguson 1982], a clause is shown in Figure 3 as a lower semi-circle (head of the clause) joined with other upper semi-circles (calls in the body of the clause) below it. During execution, the unification of a call (an upper semi-circle) with the head of a clause (a lower semi-circle) is indicated by juxtaposing the the two to form a completed circle. Figure 3 shows the result of executing a simple grandparent program. Completed circles are seen for calls to 'grandparent', and 'parent' and 'dad' twice each. The unit clauses for 'dad' are shown as single lower semi-circles, and the top-level goal as an upper semi-circle with no parent. Parameters are included as part of the semi-circles.

Such a diagram provides a snapshot of a single solution, and the successful unifications which make it up. It may thus be considered as an and-tree with the individual nodes as the goals and sub-goals satisfied. Such a display gives no indication of the steps used to get to the solution. One approach to this would be to display a full and-or tree with all the alternative solutions attempted and rejected so far. However, such a display is visually confusing and presents considerable problems when attempting to show the bindings of variables. Figure 4 shows the compromise used in DEWLAP. The program illustrated is the same as in Figure 3, and the steps in building up the display are shown. Below each call, all the clauses which could potentially unify with it are shown bracketed within diagonal lines. There are two clauses for 'parent' and these are shown adjacent to each other below each 'parent' call. The large number of unit 'dad' clauses are shown beneath each 'dad' call. At any time only one alternate clause will be successfully unified. The others either will have been tried and failed or will not yet have been tried. Clauses which have been tried and have failed are crossed out (see unit clauses for 'dad' at bottom right of Figure 4d). Only the currently unified clause is further expanded at lower levels of the tree. To further indicate the status of the successfully unified clauses, the calls are shown adorned with one of four indicators: inward pointing arrows — currently executing, that is unsatisfied goals remain below this call; triangles — backtracking is occurring; squares — success, this goal and all its sub-goals have been satisfied; unadorned — no execution has yet been attempted.

When there are a large number of clauses which might satisfy a given goal, it becomes necessary to scale the components in the lower portion of the display so that they will not overlap. This results in a display in which the high-level aspects of the execution are emphasized, while lower-level components fade into obscurity.

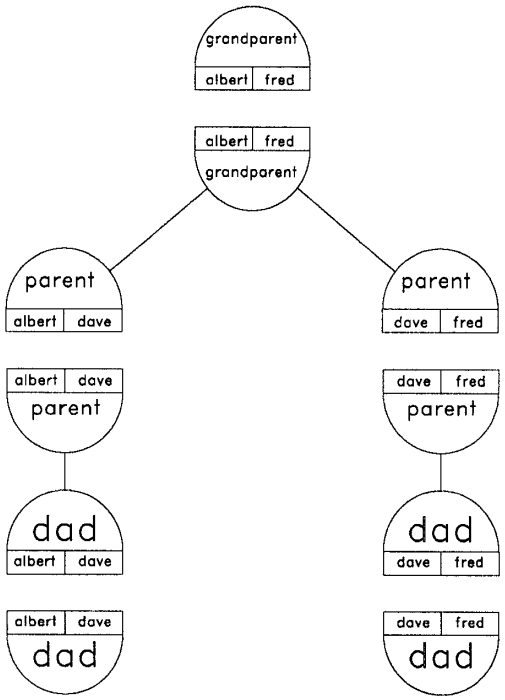


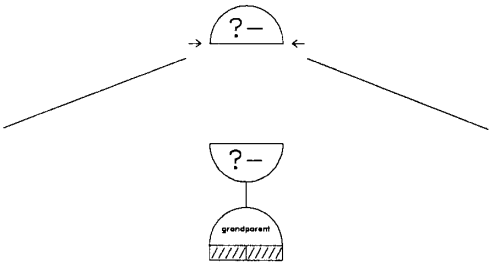
Figure 3. Prolog clauses as Ferguson diagram

Thus, the user is given an inherently high-level view of the algorithm. If it is necessary to examine low-level components, zooming and panning of the display may be done, allowing concentration on the relevant details while forgoing the high-level overview. For example, the parameters to a call are scaled to fit within the semi-circle for the call (the details of data-structure display are given below), so the number of parameters and their general form are readily available while examination of fine detail requires zooming.

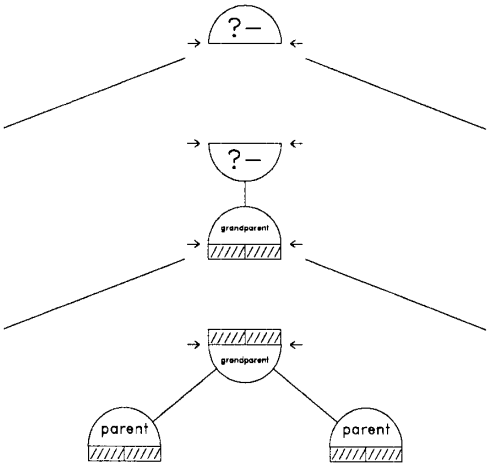
A particularly convenient form of zooming is illustrated in Figure 5. A menu selection is made for the zoom operation and a node in the display is selected using the mouse. The display is then recreated with the selected node and its sub-tree rescaled so that they just fit the screen space available. The figure also illustrates the Jade environment within which DEWLAP runs. This provides context-sensitive menus with associated help displays. In the figure, the "zoom node" option in the menu is selected and help has been requested for this command (help is generated dependent on the current position of the cursor, the current menu, and any menu item selected).

Another advantage of the graphical tree display is that logically closely related calls occur close together in the display. For example, the currently active calls (those with unsatisfied sub-goals) can be found by tracing down one branch of the tree from the root following nodes annotated with arrows.

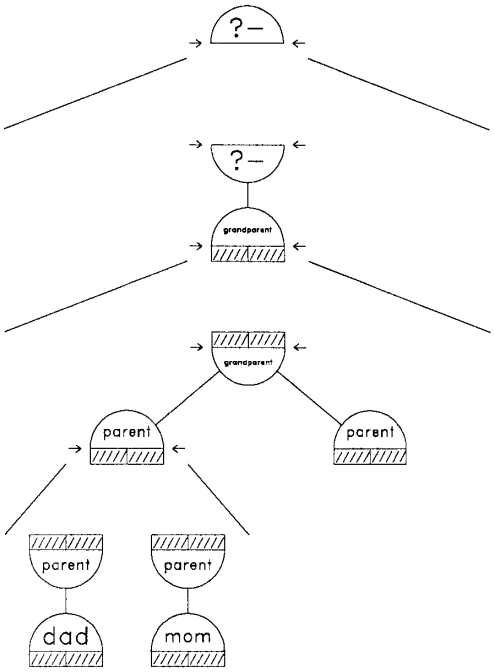
One consequence of this is that control flow moves in small local steps through the tree. For example, when a call succeeds, the current focus of execution will either move across the display one step to a sibling node, or at worst move a number of steps up the tree until an unsatisfied sibling goal is found. Similarly, during backtracking, failure of a goal causes control to move to the left and down. The importance of this is that these moves are visually simple steps, whereas in the linear context of a textual debugger they can cause dizzying jumps of context. This can be seen in a simple form in the program of Figure 4. Failure of the second 'parent' goal causes control to move back to the 'dad' goal under the first 'parent'. Thus, failure of a high-level goal



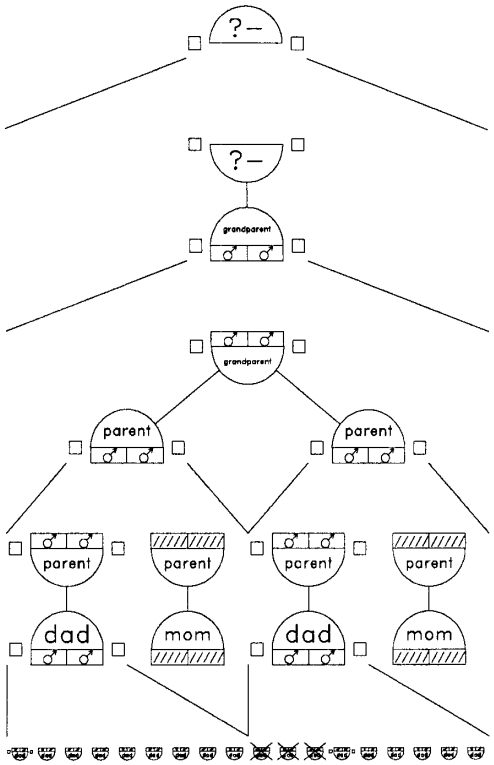
(a) top-level goal



(b) before 'parent' encountered

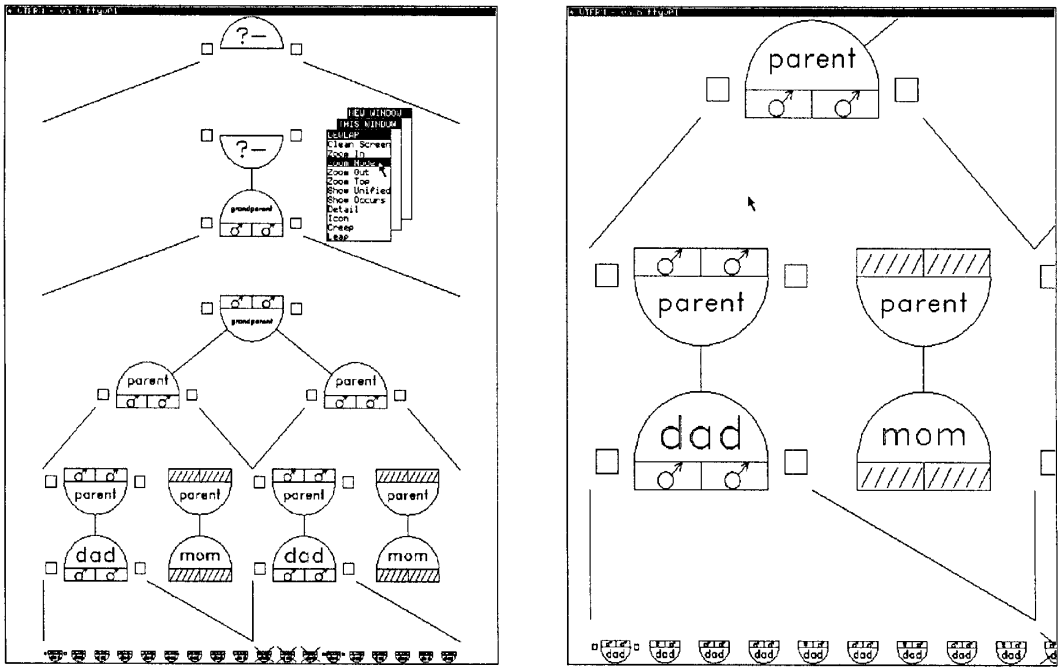


(c) after 'parent' encountered



(d) after solution found

Figure 4. And-or representation of an execution



(a) making the menu selection

(b) the display after zooming

Figure 5. Zooming on an intermediate node

can cause transfer to a deeply-nested call. In a linear trace, it is very difficult to establish the higher-level context of this low-level goal, as the calls corresponding to the earlier appearance of the high-level goal can be an arbitrary distance back in the listing. In the graphical tree display, the next goal to be retried in backtracking can be found by moving one step left to find the sibling high-level goal, and moving down the right-most branch of its sub-tree to find the low-level goal to be retried next. This makes very plain the context of that low-level goal.

Control of execution flow

An important feature of debuggers is the ability to direct the control flow of the executing program. The most necessary and basic form of this is the *breakpoint*, where the debugger pauses and allows interactive control when a particular call is made. Breakpoints are not so essential in DEWLAP because it can be interrupted and commands entered at any time. For example, the zoom option can be selected while the display is being drawn and updated. However, it is still necessary to pause the output to examine the current situation. DEWLAP allows breakpoints to be set by pointing at a particular node. The break can optionally be set to occur just on that particular call, or generically on all instances of calls with the same name. This provides a very fine and precise control over breakpoints and similar options. This style of interaction almost completely removes any need for textual input to the debugger itself. The name of a predicate need only be typed in if a generic breakpoint is to be set on a predicate not currently displayed.

Previous states

One way of thinking of the DEWLAP display is as a picture of the current active and backtrack stacks. The only information given about previous states of the calculation are the lists of tried and untried clause heads listed at each goal. That is, the display shows the successful route to the current solution but not the unsuccessful side-paths taken in the process. The advantage of this is that it greatly simplifies the display and makes it easy to understand in terms of a single static structure. However, previous unsuccessful attempts may be very important for the debugging process. DEWLAP handles this by allowing the user to force the system to backtrack to a particular point and then to retry from that point. The backtrack point is selected by pointing either at a goal or at a clause head listed below a goal. The clause head could be one previously retried or one yet to be tried. This allows previous states to be recreated or execution to be forced forward to yet untried states.

3 Display of data structures

Displaying data structures in a debugger has many of the same problems as displaying the control flow. There is a lot of detail to be shown, yet the important pieces of information should not become overwhelmed and inaccessible in the mass of detail. An additional problem in Prolog, which uses the logical variable, is finding out whether two variables have been bound to each other and finding all places where a particular variable or data structure occurs. DEWLAP solves these problems by using the same style of graphical tree display used for the execution tree and by providing display options to show all occurrences of a particular variable or data structure.

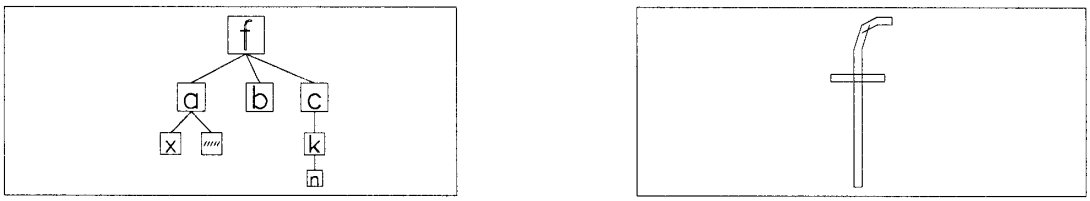
Default displays for data structures

Figure 6 shows the graphical display for a list. Each individual item in the list is scaled down in size from the previous item (in this case by 0.75). Any list will fit into a fixed length, so that complex decisions about placement within a more general display do not need to be made. But more importantly, the most important items on the list (those occurring toward its head) are displayed most prominently, with the details in the tail de-emphasized and made less visually distracting by being small, although any item in the list can be examined in detail by zooming. Items near the head are most important because they are likely to be the ones directly affecting the outcome of any unifications involving the list and so are of most concern to the user. Figure 6a shows a terminated list of four items, and 6b shows a list whose tail is an uninstantiated variable.

Figure 7 shows the display used for arbitrary (non-list) data structures. The display is a tree with the functor names occurring in the nodes. Each level of the tree is scaled with respect to the level above (in this case by 0.75), and the whole tree is scaled to fit within a box of unit size. As above with lists, this allows easy positioning of the display within a larger display, the most important items are displayed prominently with less important items de-emphasized, and any item can be examined in detail.

(a) standard list (b) variable-terminated list (c) detail omitted

Figure 6. Display format for lists



(a) detailed tree format (b) detail omitted

Figure 7. Display format for data structure f(a(x,_),b,c(k(n)))

The list and general displays used above can be very time-consuming to draw. To alleviate this problem, a simple or *icon* form of display is provided. Figure 6c shows the icon form for a list. For an arbitrary data structure the icon is the name of the top-level functor, as in Figure 7b. The iconic form is displayed by default, and to get the detail for a particular parameter in a goal node it is necessary to set the detail mode for it. (As usual, this is done by pointing at the node in question after making a menu selection.)

User-tailorable displays

The user's interpretation of a particular data structure can be very important for understanding. For example, the program in Figure 1 is transposing a two-dimensional array represented as a list of lists. An actual two-dimensional layout is much clearer than either the textual representation of Figure 1 or the list representation used by DEWLAP. To accommodate such user-dependent views, graphical routines can be written to display particular data structures at either or both the icon and detail levels. For example, in Figures 4 and 5, routines are provided for each of the individuals in the family tree. The icon representation for males is a male symbol, and for the females a female symbol. The detail level is a picture of a different face for each individual. Figure 8 shows detail taken from Figure 2, showing the two-dimensional array drawn as a rectangular array.

A high-level hierarchical graphics system called GROWL is used to provide the user interface for programming the displays [Cleary 1984]. (It was also used to program the icon- and detail-level default displays as well as a majority of the execution-tree display.)

One of the problems raised by Figure 1 is how to tell whether variables and data structures occurring in different parts of the display are the same. Figure 9 shows how this is done in DEWLAP. The user selects a menu item, "show unified", and points at the data structure or variable in question. A line is then drawn from the selected item to all other occurrences of it. This both shows occurrences on the current display and indicates any which occur out of the range of the current (possibly zoomed) window. While it is a little inconvenient to have to specifically select an item to see its occurrences, the alternative would seem to be to display similar information for all possible interrelationships at all times. The result would be an unintelligible mess of intersecting lines. Only one data structure at a time can be selected for "show unified" so this confusion is avoided.

The "show-unified" facility is implemented in such a way that data structures with the same value are not always shown to be the same. Instead, such an indication is given only if a unification has taken place between them. This is done for reasons of efficiency; keeping track of unifications is a simple matter, whereas showing all instances of identical structures would require comparing the selected parameter to all other data structures. As a result of the chosen implementation, identical structures which arise independently in different parts of the execution are not shown to be related. This is advantageous if the two instances are indeed unrelated, but somewhat misleading if they are related and a unification which will occur between them has not been executed

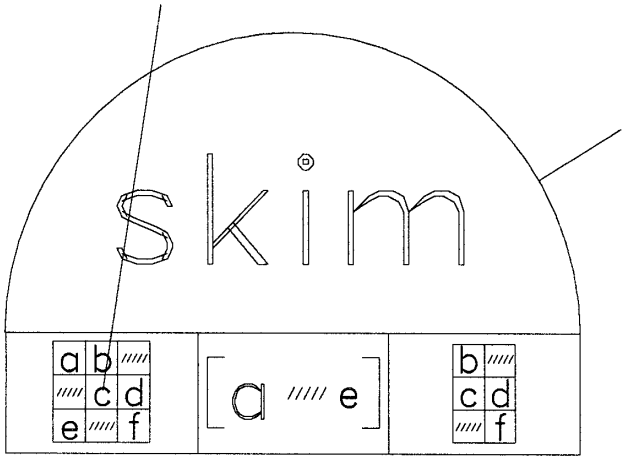


Figure 8. User-written display of array (from Figure 2)

yet. "Show unified" can also be used with an uninstantiated variable to show all variables which have been bound to it. A similar "show-occurs" facility may be used for indicating occurrences of data structures in which the selected item appears as an argument; an example of this may be seen in Figure 2.

4 Implementation

The DEWLAP system is implemented within the context of the Jade project at the University of Calgary [Witten 1983, Unger 1984]. The Jade environment provides support for development of distributed software systems, and includes a menu-driven window system, an inter-process communication facility, and graphics routines. The DEWLAP system consists of a number of communicating processes which make use of these facilities. It is currently implemented on a central Unix system running on a VAX 11/780 and a number of different remote workstations which include bit-mapped displays and mice. The user programs and the debugger itself run in C-Prolog on the VAX 11/780. The only non-Prolog routines used are the graphical driver routines provided as part of the Jade environment [Wyvill 1984]; these run both on the central machine and on the workstations. (Experiments are currently underway to migrate more of the graphics and possibly some of the Prolog itself to the workstations.) User selections of display options and breakpoints are specified with a mouse. Keyboard input is not required for the majority of the debugging task.

Environment

The standard Jade workstation is a monochrome bit-mapped 68000-based workstation, with a resolution of 560 pixels horizontal by 720 pixels vertical. Windows of arbitrary size may be created within this area. The window manager (resident on each workstation) provides a means of drawing vectors in windows, creating pop-up menus, detecting menu selections, and obtaining the coordinates of a three-button mouse. One button is used for each of menu selection, help, and pointing.

Inter-process communication in Jade is performed via Jipc [Neal 1984], a Thoth-like [Cheriton 1979] message-passing protocol. Processes may reside on any of three VAX 11/780 systems, and on any of a number of workstations.

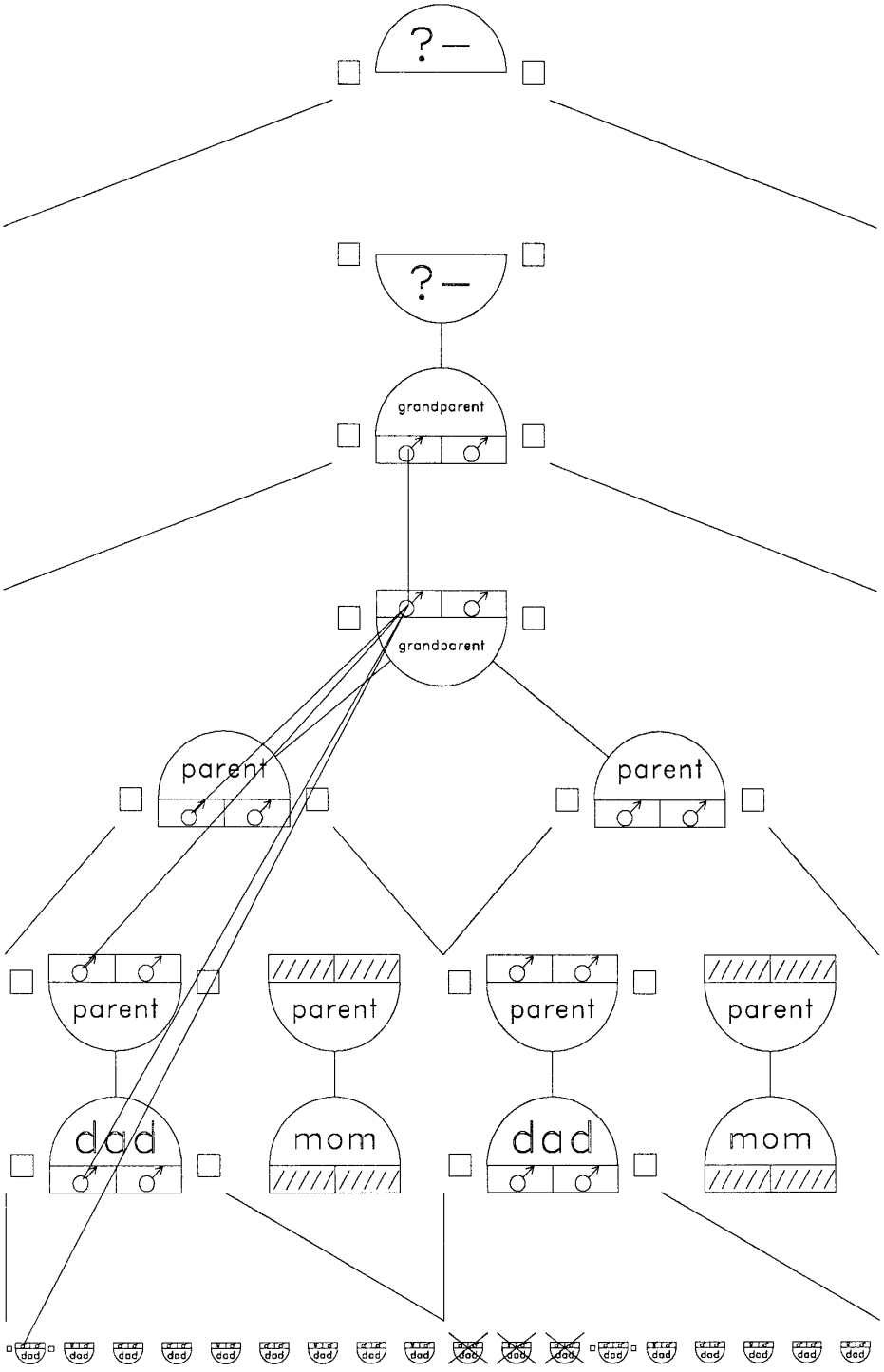


Figure 9. Indication of repeated structures

The Jade graphics system, Jaggies [Wyvill 1984], is a hierarchical vector graphics system which allows recursive definitions of pictures The GROWL graphics language used in DEWLAP is implemented on top of Jaggies. The low-level Jaggies routines are invoked remotely, via Jipc, though work is currently underway to make use of the Jaggies routines directly, within a modified version of C-Prolog [Pereira 1984].

Process distribution

The DEWLAP system consists of a number of different processes, all communicating via Jipc. The primary ones, shown in Figure 10, include a Prolog-execution process, a graphical output process, a debugger request-handling process, and a buffer process.

Interaction between the user and the Prolog program occurs in a window controlled by the Prolog-execution process. This acts like a standard login terminal to Unix. The interface is the same as the standard C-Prolog interface. Execution is performed by a Prolog interpreter (implemented on top of C-Prolog) which generates graphics requests corresponding to each execution step.

Graphical interaction occurs in a second window, controlled by a Jaggies process and an event-handling process. The Jaggies process simply accepts messages and performs the requested output or returns requested information. The event handler detects any events occurring within the graphics window, such as keystrokes, menu selections, and pressing or releasing the mouse "point" button. Menu selections are interpreted and appropriate actions performed.

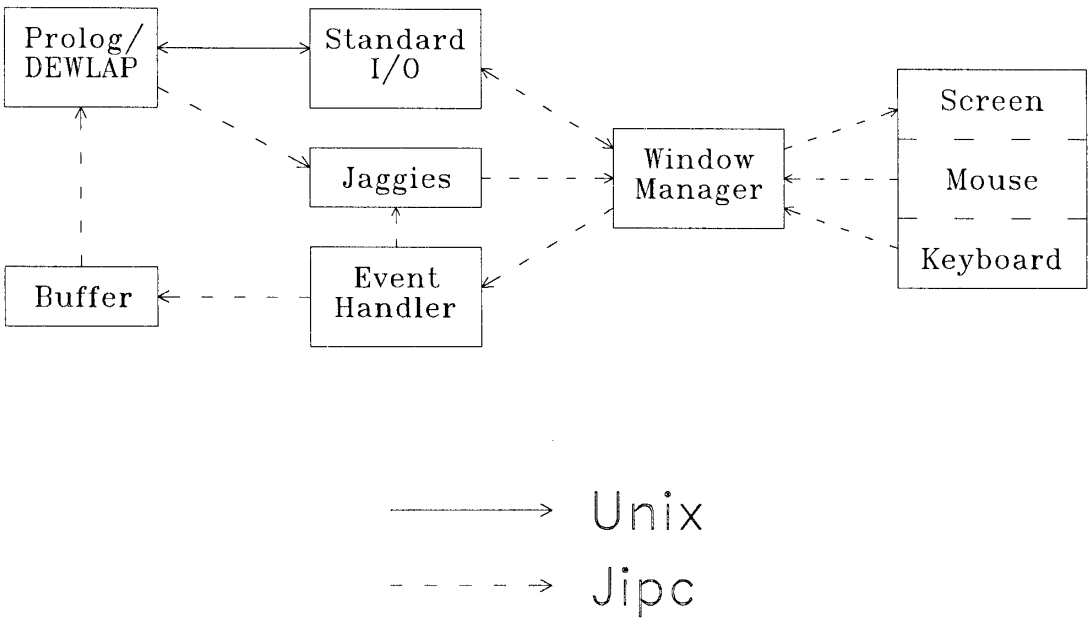


Figure 10. System diagram for DEWLAP

Execution speed

In its current incarnation, DEWLAP runs too slowly to be considered a production tool. Three factors contribute to this: the debugging interpreter is written in Prolog and runs on top of a Prolog interpreter; the construction and display of graphics introduce a high overhead; and a considerable amount of inter-process communication is required for the graphical display.

A careful re-implementation should drastically improve the speed of the system. A Prolog compiler will eliminate the second level of program interpretation. Integral graphics support will do away with much of the inter-process communication. Slow drawing of vectors may be circumvented by delaying the updating of the display until the user requests it or some member of a specified set of events occurs. These facets of the implementation were not emphasized in the initial implementation as they were considered secondary to the display issues.

5 Conclusions

One solution to the psychological problem of how to present debugger output has been described. Two-dimensional displays and the inherently simple execution model of Prolog have been combined to produce a simple and powerful debugger. The simplicity of Prolog is obscured in the usual textual debuggers where the inherently non-linear flow of control in Prolog is forced into a linear sequence.

While intuitively appealing, it is not proven that the graphical display used by DEWLAP speeds the debugging process itself. We hope in the future to do controlled tests to check this. It is also unclear to what extent the experience with Prolog can be carried over to debuggers for other languages. The hierarchical scaled graphical displays can certainly be used for displaying data structures; however, most languages do not have a simple interpretation corresponding to the execution tree used for Prolog.

Acknowledgments

We would like to acknowledge the stimulating and supportive environment provided by the members of the Jade project at the University of Calgary. Both Jade and the authors were supported by the Natural Sciences and Engineering Research Council of Canada.

References

Ron Baecker (February, 1975) "Two Systems which Produce Animated Representations of the Execution of Computer Programs" *SIGCSE Bulletin, 7* (1) 158-167.

Birtwistle, G.M., Wyvill, B.L.M., Levinson, D., and Neal, R. (February, 1984) "Visualizing a Simulation Using Animated Pictures" in *Proceedings of the SCS Conference on Simulation in Strongly Typed Languages,* pp 57-61. San Diego, California.

Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza (August, 1985) "Program Visualization: Graphical Support for Software Development" *Computer, 18* (8) 27-35.

Marc H. Brown and Steven P. Reiss (March 20-23, 1983) "Debugging In The BALSA-PECAN Integrated Environment: Position Statement" *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High-Level*

Debugging.

Marc H. Brown and Robert Sedgewick (July, 1984) "A System for Algorithm Animation" *Computer Graphics, 18* (3) 177-186.

Lawrence Byrd (1980) "Understanding the Control Flow of Prolog Programs" in *Proceeding of Logic Programming Workshop.* Debrecen, Hungary.

Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R (February 1979) "Thoth: a portable real-time operating system" *Communications of the Association for Computing Machinery, 22* (2) 105-115.

Cleary, J.G. (1984) *A distributed graphics system implemented in Prolog.* Research Report 84/173/31, Department of Computer Science, University of Calgary.

Alan Dewar and Brian Unger (February, 1984) "Graphical Tracing and Debugging of Simulations" in *Proceedings of the SCS Conference on Simulation in Strongly Typed Languages,* pp 66-73. San Diego, California.

Alan Dewar (1985) *A Graphical Debugger for Prolog.* M.Sc. Thesis, Department of Computer Science, University of Calgary.

Alan Dewar (1986) *A Graphical Debugger for Prolog.* Research Report, Department of Computer Science, University of Calgary.

M. H. van Emden and R. A. Kowalski (1976) "The Semantics of Predicate Logic as a Programming Language" *Journal of the ACM, 23* (4) 733-742.

Ronald J. Ferguson (1982) *A Prolog Interpreter for the Unix Operating System.* Master's Thesis, Department of Computer Science, University of Waterloo.

Sidney L. Getz, George Kalligiannis, and Stephen R. Schach (March, 1983) "A Very High-Level Interactive Graphical Trace for the Pascal Heap" *IEEE Transactions on Software Engineering, SE-9* (2) 179-185.

Christopher F. Herot, David Kramlich, Richard Carling, and Gretchen P. Brown (March 20-23, 1983) "Debugging in an Integrated Graphics Programming Environment" *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging.*

Joyce, J., Birtwistle, G.M., and Wyvill, B.L.M. (1984) "ANDES: An Environment for Animated Discrete Event Simulation" in *UK Simulation Conference.* Bath, U.K..

Russell B. Kline, Gary D. Hamor, Kenneth L. Krause, and Larry E. Druffel (February, 1978) "Visual Demonstration of Program Execution" *SIGCSE Bulletin, 10* (1) 16-18.

Ralph L. London and Robert A. Duisberg (August, 1985) "Animating Programs Using Smalltalk" *Computer, 18* (8) 61-71.

Brad A. Myers (July, 1983) "Incense: A System for Displaying Data Structures" *Computer Graphics, 17* (3) 115-125.

Lee Naish (1983) *MU-PROLOG 3.0 Reference Manual.* Department of Computer Science, The University of Melbourne.

I. Nassi and B. Shneiderman (1983) "Flowchart techniques for structured programming" *ACM SIGPLAN Notices, 8* (8) 12-26.

Neal, R., Lomow, G.A., Peterson, M., Unger, B.W., and Witten, I.H. (May 1984) "Experience with an inter-process communication protocol in a distributed programming environment" in *Proceedings of the Canadian Information Processing Society Session '84.* Calgary, Alberta.

Fernando Pereira (1984) *C-Prolog User's Manual.* SRI International, Menlo Park, California.

Luis Moniz Pereira, Fernando C.N. Pereira, and David H.D. Warren (1978) "Users's Guide to DECsystem-10 Prolog" D.A.I. Occasional Paper No. 15, Department of Artificial Intelligence, University of Edinburgh.

M. C. Pong and N. Ng (1983) "PIGS--A System for Programming with Interactive Graphical Support" *Software--Practice and Experience, 13,* 847-855.

S. R. Schach (1982) "An Interactive Graphical Array Trace" *Quaestiones Informaticae, 2* (1) 23-26.

Stuart C. Shapiro and Douglas P. Witmer (February, 1974) "Interactive Visual Simulators for Beginning Programming Students" *SIGCSE Bulletin, 6* (1) 11-14.

Libor A. Spacek (Summer, 1982) "A Portable Prolog Tracing Package" *Logic Programming Newsletter,* 7-9, Portugal.

Mark Stefik, Daniel G. Bobrow, Sanjay Mittal, and Lynn Conway (Fall, 1983) "Knowledge Programming in LOOPS: Report on an Experimental Course" *The AI Magazine,* 3-13.

K. Takahashi, T. Aso, and M. Kobayashi (1982) "Visual Aid for Fortran Program Debugging" in *Proceedings of the Sixth International Conference on Software Engineering,* pp 414-415. Tokyo, Japan.

Tim Teitelbaum and Thomas Reps (September 1981) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment" *Communications of the ACM, 24* (9) 563-573.

Unger, B., Birtwistle, G., Cleary, J., Hill, D., Lomow, G., Neal, R., Peterson, M., Witten, I., and Wyvill, B. (February, 1984) "Jade: a simulation and software prototyping environment" in *Proceedings of the SCS Conference on Simulation in Strongly Typed Languages,* pp 66-73. San Diego, California.

Raymond Weisling (September, 1981) "Tracemark, An Apple II Debugging Aid" *MICRO--The 6502/6809 Journal* (40) 79-80.

Witten, I.H., Birtwistle, G.M., Cleary, J., Hill, D.R., Levinson, D., Neal, R., Peterson, M., Unger, B.W., and Wyvill, B. (July, 1983) "Jade: a distributed software prototyping environment" *ACM Operating Systems Review, 17* (3) 10-23.

Wyvill, B.L.M., Neal, R., Levinson, D., and Bramwell, R. (May 1984) "JAGGIES--a Distributed Hierarchical Graphics System" in *Proceedings of the Canadian Information Processing Society Session '84,* pp 214-217. Calgary, Alberta.