Textual Tree (Prolog) Tracer: An Experimental Evaluation DRAFT: not to be quoted*

Mukesh J. Patel, Chris Taylor & Benedict du Boulay
School of Cognitive and Computing Sciences,
The University of Sussex,
Falmer, Brighton BN1 9QH, UK.

Email: bend@cogs.sussex.ac.uk

Abstract

We report the findings of the effect of Prolog trace outputs' format and information content on simple Prolog problem solving performance of novice Prolog programmers. In this study trace outputs based on Transparent Prolog Machine (TMP*), Spy (based on Byrd Box) and Textual Tree Tracer (TTT) are evaluated for effectiveness in providing information about Prolog program executions; the last one is a new (prototype) tracer developed at The University of Sussex which was designed to partly overcome some of the shortcoming of other tracers including those evaluated in a previous similar study (Patel, du Boulay and Taylor, 1991a). Subjects (n=13) solved simple Prolog programming problems with the aid of a trace output from one of the tracers. The results show that there was little overall difference between TTT and TPM* based trace outputs. Analysis of responses reveal relative strengths of TTT and confirm weaknesses of TMP* trace outputs observed in a similar previous study (Patel et al 1991c). Implications of similarities and differences of findings from both studies are discussed, as are issues related to the effect of format (or notation) on access to information and subjects' comprehension of Prolog.

1 Introduction

The overall aim of this experimental study, together with a much larger similar study (Patel, du Boulay & Taylor, 1991a, 1991b & 1991c), was to evaluate the usefulness of different Prolog program trace outputs for solving simple problems associated with novice debugging activity. Prolog is a complex and powerful programming language. Certain key aspects of Prolog remain *implicit* (or "hidden") during program execution. Prolog tracers are designed to provide information about hidden features such as flow of control, and therefore can be particularly useful to Prolog learners. The hidden mechanisms of Prolog can be described or explained (or traced) in more than one way with different *perspectives* emphasising different aspects, such as variable binding, flow of control, recursion, search space, etc. (Pain & Bundy, 1987). Often it is not possible to present information about all aspects both simultaneously and equally clearly. This is partly because of the nature of the language; emphasis on one aspect, such as flow of control, often precludes the possibility of emphasis on other aspects without loss of clarity. Format or notation (Gilmore 1991a), for example, textual or graphic, acts as further constraint on information representation

*This work was supported by a grant from the UK Joint Research Council Initiative in Cognitive Science/HCI. The experimental work was conducted using the POPLOG programming environment.

(which is discussed at length in Patel et al., 1991c). For example, a perspective focusing on the flow of control in program execution can be presented as an AND/OR tree either as a top down (graphic) tree (Eisenstadt & Brayshaw 1988) or a left to right sideways (textual) tree as in TTT.

Perspective and format interact with the information content in a trace output. On strictly formal basis, information on program execution cannot vary across tracer; the logical properties of Prolog ensure that tracers generally provide te *minimum* information necessary for reconstructing the whole 'story' of a program's execution, assuming one has an adequate grasp of the underlying logic and access to the source code. However, the level of detail and explicitness of information can vary between different tracers. Together, perspective, format and level of detail and explicitness determine ease of *access* to information which in turn determines their usefulness.[1].

Major factors that determine tracer outputs' usefulness are described and discussed in greater detail in a report of findings of a previous evaluative study (Patel el al, 1991c) in which three different types of *static* (see below) trace outputs were evaluated. The first was Spy (or Byrd Box), which provides basic traces with very limited explicit information in a textual format not always easy to comprehend (Byrd, 1980). The second, Transparent Prolog Machine (TPM*), is an *idealised* version (hence the '*') of a commercial product (TPM-CDL) based on the original TPM tracer designed by Eisenstadt and Brayshaw (1988) which graphically displays flow of control and backtracking as an AND/OR tree. The third, Enhanced Prolog Tracers for Beginners (EPTB), a textual tracer developed by Dichev and du Boulay (1989) designed to overcome some of the obvious shortcomings of the Spy tracer. The findings of this study show that not all these factors are equally important and that different combinations of perspective, format and level of detail determine trace outputs' relative usefulness in Prolog problem solving. Detailed analysis of the data further confirmed the correlation between ease of access and problem solving performance. For example, the adverse effect of the limitation of a graphic format AND/OR tree representation of flow of control on amount of textual detail was evident in novices' problem solving performance. The results served to emphasis the crucial tutorial role that Prolog tracers can play during the early learning stages. It also provided insights into how different aspects of perspective, format and level of detail can be blended in order to minimise the tension between them (and therefore, reduce the possibility of undermining the novices' uncertain grasp of Prolog).

Taylor, du Boulay and Patel (1991) describe a number of strengths and weaknesses of EPTB, CDL-TPM and Spy (Byrd box) tracers. Consideration of these together with some of the findings of the previous evaluative study has led to the conception of the Textual Tree tracer (TTT). It incorporates a number of the better features of existing tracers, whilst avoiding some of their shortcomings, together with some novel features. The present study was designed to evaluate the usefulness of a TTT (as compared with Spy and TPM*) trace outputs. TTT is currently under development, and exists only as a partially constructed prototype and its design features are still evolving. All three tracers are described in more detail in the next section. The overall aim was to investigate the effect of perspective, format and level of detail of Prolog trace outputs on access to information necessary for solving problems associated with simple debugging tasks.

Note that these trace outputs were evaluated as static representations in a non-interactive mode. In each case an appropriate screen dump of the relevant trace output was shown in its entirety, so users could not 'grow' the trace nor add or delete information from it. Thus, variation between method of control between tracers was eliminated. Our intention was not to examine the tracers 'in the round' but to focus on the influence of format and level of detail in *static* trace outputs based on Spy, TPM* and TTT tracers on the clarity and accessibility of information.

More specifically, this report focuses on TTT's performance *vis-a-vis* our set of stimuli problems which are similar to the ones used in the previous study. Given this similarity in the problem

[1] The task and the level of user expertise as well as user interface also play a role in determining overall usefulness of help tools such as Prolog tracers. However, the effects of these were not evaluated in this study, and have been discussed elsewhere (Patel et al. 1991c)

solving task we also expected to confirm pervious findings of the relative usefulness of TPM* and Spy trace outputs. So the results presented here are also of general methodological interest; it provides data which allows us to compare the results of this and the previous evaluative studies. The combined findings of both experiments help to clarify the nature and scope of programming help tool evaluations. Further, it is possible to interpret these findings with a great deal of confidence because it is possible to identify the reasons for differences in the findings of both studies.

1.1 Prolog Tracers

In this section Spy and TPM* Tracers are briefly described, followed by be more detailed account of TTT which is the main focus of this report.

1.2 Spy Type Trace Output

Spy is a very basic textual (linear) tool and is included in this study because subjects were familiar with it. The version used in this study did not show system goals. This tracer provides most of the basic information necessary for programming or debugging in Prolog, but much of it is implicit. In particular, the relationship between the source code and the trace output is not as clearly displayed as it is in TPM* and TTT, which are both designed to overcome some of the obvious shortcomings of Spy. In the previous evaluation study Spy trace outputs were not nearly as helpful as TPM* or EPTB in the problem solving task. In the present study it was not expected to perform any better than TTT.

1.3 TPM* Type Trace Output

The TPM* (Transparent Prolog Machine) is a tracing tool which makes use of a modified and extended AND/OR tree representation, known as the 'AORTA' representation. The TPM-CDL version is an interesting illustration of the use of graphical representation in tracers. However, the tracer suffers from a number of drawbacks which considerably limit its usefulness in practice (see Taylor et al., 1991 for more a detailed discussion). Trace outputs used in this study are based on an *idealised* version of TPM-CDL, which are free of these drawbacks evident during interactive use. There are now other versions of TPM available, particulary one for the MAC, which offer significant improvementsbover CDL-TPM. Further, the outputs were significantly modified to include all the relevant details otherwise optionally selected by the user. The spatial layout of TPM* provides a great deal of information at a glance, particularly on the flow of control and search space. A TPM* trace output also looks a lot less cluttered compared to TTT. Overall it gains in clarity by exploiting some of the advantages of graphic format outlined above. However, the display of argument instantiations makes it difficult to see the bindings that variables have obtained, particularly in the case of large data structures such as lists. This constraint is a direct consequence of the format of the tracer. The use of a graphical representation of AND/OR trees restricts the screen space available for augmenting with information about predicates with a large numbers of arguments, or variables with long names. This problem can be overcome by including a scrolling facility but the display of essentially textual information is still poor compared to more conventional textual tracers such as EPTB, TTT and Spy. The diagrams in an Open University Prolog course text (Eisenstadt & Dixon, 1988) demonstrate the strengths of TPM-style trace output.

1.4 TTT Type Trace Outputs

TTT is a textual, non-linear tracer. It uses a *sideways tree* notation relying on text rather than graphics with the 'root node' at the top left, branches growing towards the right, and new subtrees of a node being added below any previous subtrees of that node. Immediate subcalls of a call are shown indented by one character width from the left-hand edge of the trace with respect to that call. Like the EPTB, it also shows clause matching and retrying events, distinguishes several failure modes, and provides detailed more explicit information about variable bindings. The last are presented with variable names used in the program code, on a couple of lines below the relevant (numbered) call line. The results of the previous study confirms the benefit of a high degree of explicitness (at least for the present experimental task).

Also, unlike Spy, and like TPM*, the information about a particular call is *localised* in and around the line showing that call in TTT notation. In Spy notation the outcome of a call is indicated by other lines (e.g. 'exit', 'redo' or 'fail' lines) which are typically some way further down the trace, usually separated by information about intervening sub-calls and calls. Without line indentation it is difficult to match pairs of 'call' lines with a corresponding 'exit' or 'fail' lines (as confirmed by results of previous evaluative study, Patel et al. 1991c). In the TTT notation, each line showing a call includes a "status field", which given abbreviated information about the clause number (if any) matching a call, together with an indication of the successes or failures of the clause. The status fields include all relevant information of previous executions of clauses; its a record of the history of the execution of a program concisely presented on one line in tandem with clause call matching (or not) information. The status field notation used in the study distinguished several different kinds of failure — for example, Fb for failure on backtracking, and Fs for initial failure of a system goal. The notation also distinguished success of a rule, that is, a clause with some subgoals (Sr), from success of a fact, that is a clause with no subgoals (Sf), and success (Ss) respectively.

In comparison with TTT notation, TPM*'s graphic AND/OR tree representation of the overall structure of the computation is more clear. However, in practice for non-trivial sizes programs the graphics take up too much display space severely limiting presentation of other relevant (textual) information. Normally, extra information is displayed in separate subwindows which is not convenient for information about calls to recursive list-processing procedures with long lists. TTT's sideways tree representation is intended to make it easier to correlate the trace output with the program clauses, and also allows more space in which to display the arguments of calls, whilst retaining the structural clarity that a tree representation provides.

At the time of the experiment the tracer had not been fully implemented. So trace outputs were constructed by hand, but given the static nature of the task this difference is of no significance to findings reported here. As regards TTT trace outputs evaluated in this experiment, their chief weaknesses are their cluttered look. Also traces are typically lengthy as a result of more detailed information described above. The elongated nature of its notation makes it difficult to clearly perceive the *overall structure* of a computation and the flow of control, particularly when backtracking is involved. In the final implementation of the TTT, considerable use will be made of default restrictions to curb the amount of trace output produced, with further information being shown only on request, so that traces will typically be kept very short, and bugs should be quickly located by a breadth-first top-down search of the trace tree. Indeed, the compactness of the TTT's trace output will be one of its most useful features. Overall, TTT traces will be shorter than Spy traces — particularly when backtracking is involved — and yet be far more informative. However, this advantage is unapparent in the present non-interactive study, in which only complete static traces were evaluated, and in which the degree of detail shown was much greater than would be usual in normal default mode operation.

2 Information, Format and Experimental Task

Apart from the effect of perspective, how do trace outputs vary in terms of overall information content? In this context, term *information* is used in a specific way; it refers to information about when, how and which clauses are matched, how variables are bound to (and unbound from) values at particular points, and the overall flow of control, including backtracking, together with the success or failure of goals. Information about the operations of a Prolog program, that is, the states it passes through, the variable bindings at each important step and the amount of backtracking involved, is useful in understanding and debugging Prolog programs. Trace outputs are designed to provide this information though because of Prolog's complexity they can vary in terms of the exact nature of information provided. The variation can be due to the level of detail. For example, Spy trace does not indicate which clause of a predicate is being used at any point, whereas TTT and TPM* do. Spy refers to program variables by their internal names such as '_405', while CDL-TPM systematically labels variables with letters, and unlike both, TTT uses the names chosen by the programmer appended with a numerical subscript to distinguish between copies. Though all three methods serve the same function, they are not equally efficient in providing relevant information for the sorts of problems that were used in this evaluative study.

All three tracers have a different emphasis on perspective, though this is not assumed to have any significant effect on usefulness on our experimental task. The basic perspective mediated by format of trace outputs affects their relative informativeness. Though it is never very easy to assess the precise effect of differences due to perspective. So it was assumed that the different perspective of each tracer provided the minimum level of information necessary to solve simple problems typically encountered in a Prolog programming task. For the purpose of this study it was assumed that the three tracers provide the minimal — and in the case of Spy this could be very minimal indeed — information, and that any main differences in their usefulness is due to format and *access* to information. More realistically, it is obvious that in most cases there would be some interaction between format and information content, and so any explanation of helpfulness of tracers would have to give an account of such an interaction. But our strong assumption of information equivalence is justified because the stimuli problems were designed and tested to ensure a fair evaluation of similar features of each tracer's judged useful for well-defined specific tasks. Further, the same rigour in designing the task material enabled us to clearly pinpoint the source of such interaction. Overall, we assume that apart from the perspective, the two main determiners of differences between tracers are format and the level of detail and explicitness of information about executed programs.

Further, trace outputs have different degree of explicit information. For example, information about number of sub goals of a clause can be highly implicit, as in a Spy trace, or fairly explicit, as in TPM* (as long the clause succeeds), and in TTT (independent of whether the clause succeeds or not). While, it is not difficult to provide exemplars to define our notion of level of detail and explicitness of information, in reality these two aspects are often closedly inter-related. However, this is not a serious drawback as long as it is clear that whatever level of detail and the degree of information explicitness, its the effect on access that determines a trace output's usefulness. Thus, it follows that simply having more detailed or explicit information does not increase usefulness, because too much detail can be hindrance in some cases. This tension between being explicit and overwhelming the user with "unnecessary" details (or redundant information) is an important determiner of ease of access to information and therefore tracer usefulness. Hence, one of the questions that this study addresses is the amount and sort of information that is useful to novice programmers.

To recap, assuming that the information content of tracers was similar for all relevant aspects of Prolog, but that they varied in terms of access to information, how would this affect users' ability to solve Prolog problems? Leaving aside the effect of perspective on information access,

it is probable that the more explicit the information, the quicker a subject would be able locate it. In this study, the task required subjects to study trace outputs in order to solve problems. Typically, they would have to work out whether a particular clause was matched on not or whether it affected the execution of another part of the program. These problems required the subjects to make inferences based on trace output. It was assumed that the more explicit the relevant information, the fewer inferences necessary, and therefore less time spent on solving the problem. However, according to this line of argument there would be no reason for ease of access to information to affect *accuracy* of response; all things being equal, a subject would be able to solve the same problem with different trace outputs though it might take her varying amounts of time. Therefore, any significant differences in response accuracy would have to be accounted for in terms of the effect of format and perspective. Such an account would strongly suggest that the choice of format and perspective had important implications for human cognitive processes.

3 Method

3.1 Design

The trace output evaluation task was presented on a VDU as part of an automated process including a learning and a trial phase. The stimuli consisted of five problems, divided into two groups which were roughly determined by aspects of Prolog on which their solutions depended. Group 1 included three very simple problems which could be solved with information on backtracking, clauses tried and undefined predicates. By contrast, solution to the remaining two problems in the second group depended (to a certain extent) on information about recursion, system goals, goals with variables, and list manipulation. An example of problems from each group is given in the Appendix. Each problem was presented three times; once with each trace output. Care was taken to disguise similarity between question across different trace output types as is evident from the appended examples. This was done by altering words and phrases of problems, as well as program code names of definitions and variables. The disguised problems were tested in a pilot study before being selected for the problem stimuli set.

All subjects were given detailed instructions on TTT and TPM* type trace outputs after which they had to solve at least 9 out of 11 problems designed to check their comprehension of these trace outputs. Subjects who were not able to reach the criterion of correct responses were excluded from the main evaluation study. The experiment was a single subject design: All subjects attempted to solve all the stimuli problems. Problems were presented in a psuedo random order; no problem was allowed to be followed by another of the same type but with a different trace. Subjects responded by picking a response from a multiple choice responses. Data on time taken to solve a problem as well as the chosen response were collected.

3.2 Subjects

13 undergraduate novice Prolog programmers at Sussex University *completed* the problem solving task, and were paid five pounds for taking part in both parts of the experiment.

3.3 Procedure

The instructions and problem solving task were presented on Sun workstations in three stages. The entire process was self-administered by the subject, who reponded by pressing a few keys on the keyboard. Following the preliminary instructions explaining the aim of the study and the nature of the problem solving task, subjects were given a tutorial on non-interactive, modified trace outputs based on TPM* and TTT tracers. Descriptions of various features of both type

Problem	Group 1			Group 2		
Trace	1	2	3	4	5	Mean
TPM*	52.7	69.7	67.9	74.5	155.0	84.0
Spy	66.4	115.0	55.8	109.4	152.4	100.0
TTT	52.1	85.5	115.5	64.0	170.5	97.5
Mean	57.1	90.4	79.7	82.6	159.3	-

Table 1: Times (secs.) of all Problem by Trace (n=13)

of trace outputs assumed a basic understanding of Spy traces outputs. Subjects who felt that they had an inadequate understanding of Spy tracers therefore did not take any further part in the study. In the next stage subjects were required to pass a criterion test designed to ensure that they had the necessary understanding of TPM* and TTT to be able to attempt solving the problems. The criterion test had 11 questions on various features of both tracers, and subjects had to get at least 9 correct in order to proceed to main part of the experimental task. Subjects were allowed three attempts to reach this criterion. Following an incorrect response, subjects were given feedback explanations of the correct response. Those who failed to meet to criterion did not take part in the rest of the study. This procedure ensured that only subjects with an adequate understanding of Prolog as well as TPM* and TTT were included in the results reported here.

The third stage was the main problem solving task. Each problem was presented as a multiple choice question (with the order of choices randomised) which the subjects were requested to read through before pressing a key to see the accompanying text. This enabled us to collect data on time spent reading the question separately from time spent trying to solve the problem with the aid of a trace output. Subjects picked a response which they had to confirm by pressing an appropriate key which ensured the possibility of altering unintended responses. Response data were recorded, but subjects were given no feedback on them. The order of presentation was as random as possible and avoided presenting the same question (but with different trace outouts) consecutively. Subjects were asked to complete the task as fast and as accurately as possible.

3.4 Results

Solution times (ST) ANOVA was carried out with subjects as the random factor and Trace Output (3 levels) and Problem (5 levels) as fixed factors. All solution time data is included in the analysis. There was no significant main effect of trace output, indicating that differences in overall mean solutions times are not significantly affected by tracer type (see Table 1). There is a significant main effect of problems, $F(4,48) = 20.96$, $p \leq 0.001$. Given the variance in level of difficulty of problems this effect was expected; subjects took varying length of time attempting to solve different problems. Broadly speaking, Group 1 problems (1, 2 and 3) took less time to solve than Group 2 problems (4 and 5).

The interaction between problem and trace type was marginally significant, $F(8,96) = 1.85$, $p \leq 0.1$. Overall, the combined effect of trace output and problem displays no particular trend. The effect of difference in format of trace outputs is not consistent across different questions. Table 2 shows solution times of correctly solved problems. As would be expected the means are generally higher but no different in trend from those based on all solution times (given in Table 1). Correct solution to problems presented with Spy traces take the longest in all expect one problem; however, unlike the rest, problem 2 is solved fasted with the Spy trace). Compared to TPM*, solutions to problems presented with TTT trace outputs take longer in problems 1, 2, 3 and 5. We will return to a more detailed analysis of these effects after presenting the response data analysis of variance.

An ANOVA similar to that of solution time data was carried out on the responses themselves.

Problem	Group 1			Group 2		
Trace	1	2	3	4	5	Mean
TPM*	54.3	76.9	72.5	83.0	161.4	85.9
Spy	66.4	153.7	59.6	171.2	193.7	97.3
TTT	57.5	106.2	119.8	72.9	162.2	101.1
Mean	59.5	105.8	87.5	81.7	166.8	-

Table 2: Times (secs.) of correctly solved Problem by Trace (n=13)

Problem	Group 1			Group 2		
Trace	1	2	3	4	5	Mean
TPM*	85	77	85	46	62	71
Spy	100	46	62	8	23	48
TTT	77	54	92	85	62	74
Mean	87	59	78	46	49	-

Table 3: Percentage Correct Response by Problem and Trace (n=13)

There is a significant main effect of problems, $F(4,48) = 8.43$, $p \leq 0.001$, which corresponds to differences in solution times reflecting the varying level of difficulty of problems. On average subjects found Group 1 problems easier to solve. There is also a significant main effect of tracers, $F(2,12) = 14.00$, $p \leq 0.001$. Overall subjects performed best with TTT (closely followed by TPM*) and worst with Spy, as show in Table 3.

There is a significant interaction between tracer and problem, $F(8,96) = 3.08$, $p \leq 0.01$. Apart from problem 1, correct responses varied significantly according to the accompanying trace output. The pattern of differences between problems solved with Spy traces is similar to that observed in the earlier study (Patel et al., 1991a and 1991b) where it was evaluated in comparision with TPM* and another textual (non-tree) Prolog tracer, EPTB (Extended Prolog Tracer for Beginners); the same is true for TPM* except that these percentages are consistently lower than those observed in a previous study. These similarities suggest that the effect of differences in trace format are independent of the specific experimental task and consistent across different combinations of Prolog trace outputs.

Apart from problem 1, Spy trace outputs are the least helpful in solving these problems; the above average mean solutions times do not seem to aid correct solutions. Even when trying hard, subjects encounter difficulties in solving problems (particularly, problem 4) with Spy. Problems 1 and 2 are solved by more subjects with TPM* traces than with TTT; the reverse is the case for the remaining problems. This is quite interesting because they both emphasise the same Prolog perspective but in different formats. Though subjects take longer to solve problems 3 and 5 with TTT traces (compared to TPM*), the responses are more likely to be correct (unlike Spy). Next we consider differences in solution times and correct responses in more detail.

3.5 Details of Trace Output and Problem Interaction

Here we attempt to account for these differences by relating each problem with information availability or access in each trace. Essentially, the following will highlight the nature of compatibility of problems with trace outputs. Without doubt the problems (and the experimental design) could not possibly have tested all aspects of each tracer adequately. Apart from the complexity of such a task, our experimental design precluded any interactive assessment. The following is a diagnostic analysis aimed at a more descriptive account of the nature of the interaction between problems and traces outputs; in particular it aims to provide a more detailed explanation of TTT trace

outputs effect on problem solving performance. Similarly detailed accounts of the effects of TPM* and Spy were reported in a previous study (Patel *et al* 1991c) and therefore will not repeated here except where appropriate in illuminating our focus on the effect TTT format and perspective on problem solving performance.

Problem 1: To solve this simple problem, subjects had to work out how often a particular procedure is called. To ensure that subjects used the trace output, it was presented without the program. Correct response were relatively high for all trace outputs. Spy traces scored better than either the TPM* or TTT probably because of explicit reference to "call" whenever the relevant clauses are called during execution. This information was no less clear in TPM* or TTT except that there are no explicit references to "calls" in either. Further, in the TTT format calls to procedures may have been confused with clauses displayed for the same procedure; extra information not available in Spy traces. More detailed notations in TPM* and TTT traces may be a hindrance for solving this type of simple problem.

Problem 2: This problem, also presented without the program used to generate the traces, required subjects to find out how many times a subgoal of a particular clause had been called. Again not a very difficult problem but one that serves to highlight one major shortcoming of Spy traces; the lack of explicit information about the number of calls to subgoals (see Patel et al. 1991c). Though the mean solution times of TPM* and TTT traces were similar, TTT trace was less helpful in solving this problem. It seems that the layout of information about clauses, clause numbers and calls to subgoals is potentially confusing in the TTT format. This may have led to miscounting subgoals which accounts for most of the errors.

Problem 3: To solve this problem subjects had to pick a false statement from a choice of four. The problem was presented without the program and with traces of the same program as for Problems 1 and 2. The correct answer was that it was false that a particular procedure succeeds. Other options (all true) included whether the first clause of a particular procedure had more than two subgoals, whether the second clause of the same procedure had exactly three subgoals, and whether a particular clause was tried. TTT and TPM* format traces were better than Spy at enabling subjects to solve this problem. However, TTT required nearly twice as long as TPM* trace outputs. The main reason being that to a generally cluttered format together with the unclear layout of TTT seems to more time to verify the true options and confirm the false one. Spy's poor performance is accounted for by its less explicit representation of information about clauses, and its non-localised display of the outcome (that is, success or failure) of calls[2].

Problem 4: This problem was presented with the program clauses corresponding to traces. To solve it subjects had to work out the number of time a clause of a particular procedure had been invoked. Trace outputs based on TTT format performed better than TPM* format in helping the solve this problem. TTT's significantly better performance reflects the clarity of its representation of this information; the status (goal) line against the relevant clause number provides a summary of every invocation of the clause (number), which is all that is necessary to solve this problem. Unlike solutions to problems 1 and 2, the confusing representation of information about calls to subgoals does not have an adverse effect on problem solving performance. The main reason for TPM* traces' high error score is the less explicit nature of some clause matching; in our static traces only the most recent clause number which matched a call is shown explicitly, and, Spy

[2]In the Spy format, the outcome of a call is shown by means of an "Exit" or "Fail" line, which may be some way further down the trace than the corresponding "Call" line, separated by several lines pertaining to subgoals, subgoals of subgoals, and so on.

performed as badly as expected because of its highly implicit representation of clause matching information. (For a detailed discussion of both these issues see, Patel et al., 1991c).

Problem 5: To solve this problem subjects had to work out from the trace how often a clause of a particular procedure was invoked. The problem was presented with the relevant program code. Solution with a TTT trace output, though more accurate, took slightly longer than that with TPM* traces. The overall lower means reflect the respective shortcomings of both types of trace outputs. As in the case of problem 4, TPM* traces obscure information about previous invocations. And the general clutter and its consequent potential for confusion in the TTT format partly accounts for the overall below average performance. Correct solutions with a Spy trace output took the longest and is the least accurate; once again the main reason being the implicit representation of information about clause matching.

4 Discussion

Broadly speaking the limitations of perspective and format (notation) of Spy tracer together with the noted spareness of explicit information about certain key aspects of Prolog is once again evident from findings reported here. The overall pattern of differences in solution times and response errors for Spy trace outputs were similar to those observed in the previous study (Patel et al., 1991c). Similarly, bearing in mind that this study involved a much smaller group of subjects, TPM* trace results more or less replicate the trends observed in the previous studies. However there are some notable dissimilarities which we presume are due to individual differences and therefore not indicative about any general properties of the trace output. The rest of discussion will be confined to TTT trace outputs' performance in this evaluative task.

Compared to EPTB's above average performance in the previous evaluative study, TTT trace outputs did not perform nearly as well. From the detailed analysis the reason for this outcome is not very clear. For example, it is not possible to explain the errors in the same way as TPM* ones can be, that is the resulting confusion due to the graphic format representation obscuring 'historical' information about program execution (equivalent to the status field in TTT traces). More generally, it seem that the particular order in which information about clause numbering, clause calls and variable binding was not very clear. While solutions to all problems relied on this sort of information, it was not possible to pinpoint the exact effect of this cluttered representation (with its potential for confusion) on the basis of response results to a particular question. But comparing the differences in performance of EPTB and TTT traces seems to suggest that some of the information in TTT format trace output was not very clear. As in EPTB, TTT traces explicitly showed the syntactic form of each called clause and the corresponding instantiation of the clause head matching against a goal and the resultant variable bindings. However, the TTT notation was perhaps less clear than the EPTB notation, one reason being that the number identifying a clause was given on a line preceding the one with the information about the relevant clause. Further, we suspect that the inclusion of a lot of details — again not a serious problem in EPTB trace output notations though it resulted in longer solution times — was superfluous for the problem solving task and may have ended up being a hindrance. So the advantage of localised information on particular clauses was dissipated by the lack of clarity in the TTT trace outputs. [3] However, there is no direct evidence of this in response data reported here.

With this in mind the latest version a prototype TTT tracer has undergone a number of changes since it was experimentally evaluated. The trace notation in the latest version is a lot

[3]At the time of the experiment, the prototype version had not been implemented. So trace outputs were artificially constructed which accounted for part of the cluttered look and 'feel' of the notation, and inevitably some errors crept in though their effect on performance was negligible since solutions to none of the stimuli problems depended on such errors in the trace.

simpler, clearer and concise (compact). Many of this improvements have been based on the experimental findings of both evaluative study. In the final version, the program code clauses will be displayed in a separate *database window*, which will include explicit clause numbers. It will also provide details on calls to subgoals (if any) of each clause suitably indented to coincide with indentation of lines in the main trace. Thus the clutter due to details about clause matches etc., will be shifted into a separate window.

In retrospect it was also felt that the status field may contain too much detail with the potential to mislead a novice user. The use of S and s or F and f for distinct meanings is potentially confusing. In particular, the combination Sf can be very easily misinterpreted as meaning "success followed by a failure on backtracking" instead of its correct meaning, " success of a fact". The notation has been made simpler and clearer by reducing the number of failure modes, and dropping the distinctions between success modes. Thus replacing Ss or Sr with an S, as the information conveyed by the smaller letters can be easily inferred from the presence (or absence) of a subtree corresponding to a clause.

5 References

Byrd, L. (1980). Understanding the control flow of Prolog programs in Tarnlund S. ed. *Proceedings of the Logic Programming Workshop*, 127-138.

Dichev, C., and du Boulay, J.B.H. (1989). An Enhanced Trace Tool for Prolog. In *Proceedings of the Third International Conference, Children in the Information Age*. 149-163. Sofia, Bulgaria.

du Boulay, J.B.H, Patel, M.J. and Taylor, C. (forthcoming). Programming Environments for Novices. To appear in *Proceedings of NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming*, Santa Margherita, Genoa, Italy, March 1992.

Eisenstadt, M. and Brayshaw, M. (1988). The transparent Prolog machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming* 5(4):277-342.

Eisenstadt, M. and Dixon, M. (1988). *Intensive Prolog: workbook*, The Open University Press, Milton Keynes.

Gilmore, D.J. (1991a). Does The Notation Matter? *mss*, Dept. of Psychology, University of Nottingham, UK.

Gilmore, D.J. (1991b). Models of Debugging. *mss*, Dept. of Psychology, University of Nottingham, UK.

Green, T.R.G. (1991). Describing information artifacts with cognitive dimensions and structure maps. In *Proceedings of "HCI'91: Usability Now" Annual Conference of BCS Human-Computer Interaction Group*, eds. D. Diaper and N.V. Hammond, CUP, UK.

Hook, K., Taylor, J. and du Boulay J.B.H. (1990). Redo "Try Once And Pass": the influence of complexity and graphical notation on novices' understanding of Prolog. *Instructional Science* 19 (4-5):337-360.

Pain, H. and Bundy, A. (1987). What stories should we tell novice Prolog programmers? In Hawley, R. ed. *Artificial Intelligence Programming Environments*. Ellis Horwood.

Patel, M.J., du Boulay, J.B.H. and Taylor, C. (1991a). Prolog Tracers and Information Access. In *Proceedings of The First Moscow HCI '91 Workshop, Moscow*.

Patel, M.J., du Boulay, J.B.H. and Taylor, C. (1991b). Effect of Format on Information and Problem Solving. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society, Chicago*.

Patel, M.J., du Boulay, J.B.H. and Taylor, C. (1991c). Working Title: Evaluation of Prolog Trace Outputs. *mss* School of Cognitive and Computing Sciences, University of Sussex, UK. (in preparation).

Patel, M.J., du Boulay, J.B.H. and Taylor, C. (1991d). Aptitude Paper reference

Taylor, C., du Boulay, J.B.H., and Patel, M.J. (1991). Outline Proposal for a Prolog "Textual Tree Tracer" (TTT), Cognitive Sciences Research Paper-177, School of Cognitive and Computing Sciences, The University of Sussex, UK.

Shu, N. C. (1988) Visual Programming. New York: Van Nostrand Reinhold.

6 Appendix

This section includes Spy, TPM* and TTT questions and screen dumps for question 2 and question 4. There three examples of each question, one for each tracer output type. In each case the question is shown together with the answer choices. The correct answer choice is starred.

```
TTT Question 2

The picture of the trace shows the output for the goal

?- z7.

for a program which contains several simple rules,
such as for example

n :- k9, w, i8.

Which one of the following statements is TRUE?

Clause 1 of w has two subgoals*
Clause 1 of n has five subgoals
Clause 2 of n has five subgoals
z7 has only one clause, which is recursive
```

SPY Question 2

The picture of the trace shows the output for the goal

```
?- e.
```

for a program which contains several simple rules of
which the following is a single example

```
g :- j, k, l.
```

Which one of the following statements is TRUE?

The first clause for k has two subgoals*
The first clause for g has five subgoals
The second clause for g has five subgoals
e is defined by a single recursive clause

TPM* Question 2

The picture of the trace shows the output for the goal

```
?- a.
```

for a program which contains several simple rules of
which the following is a single example

```
c :- f, g, h.
```

Which one of the following statements is TRUE?

The first clause for g has two subgoals*
The first clause for c has five subgoals
The second clause for c has five subgoals
a is defined by a single recursive clause

TTT Question 4

The procedure durl is defined as follows. What is the number
of times that its 4th clause is invoked (whether
successfully or not) when the following goal is evaluated:

```
?- durl([1,19], [0,1,11,21], A).

1  durl([], _, []).
2  durl(_, [], []).
3  durl([E|R1], [E|R2], [E|R3]):-
     durl(R1, R2, R3).
4  durl([E1|R1], [E2|R2], R3):-
     E1 < E2,
     durl(R1, [E2|R2], R3).
5  durl([E1|R1], [E2|R2], R3):-
     E1 > E2,
     durl([E1|R1], R2, R3).
```

Once
Thrice*
Four times
Never

SPY Question 4

Suppose the goal

```
?- trundle([7,11], [5,7,10,12], L).
```

is evaluated against the program

```
trundle([], _, []).
trundle(_, [], []).
trundle([X|Xs], [X|Ys], [X|Zs]):-
    trundle(Xs, Ys, Zs).
trundle([X|Xs], [Y|Ys], Zs):-
    X < Y, trundle(Xs, [Y|Ys], Zs).
trundle([X|Xs], [Y|Ys], Zs):-
    X > Y, trundle([X|Xs], Ys, Zs).
```

From the trace you are shown, how many
times does the head of the 4th clause
of "trundle" match a call to "trundle"?

Once
Three times*
Four times
Not at all

TPM Question 4

Let "zwysick" be defined by the following
five clauses.

```
zwysick([], _, []).
zwysick(_, [], []).
zwysick([A|P], [A|Q], [A|R]):-
    zwysick(P, Q, R).
zwysick([A|P], [B|Q], R):-
    A < B, zwysick(P, [B|Q], R).
zwysick([A|P], [B|Q], R):-
    A > B, zwysick([A|P], Q, R).
```

What is the number of invocations
(successful or otherwise) of the
4th clause when the following
goal is computed?

```
?- zwysick([3,7], [2,3,5,8], I).
```

One
Three*
Four
None

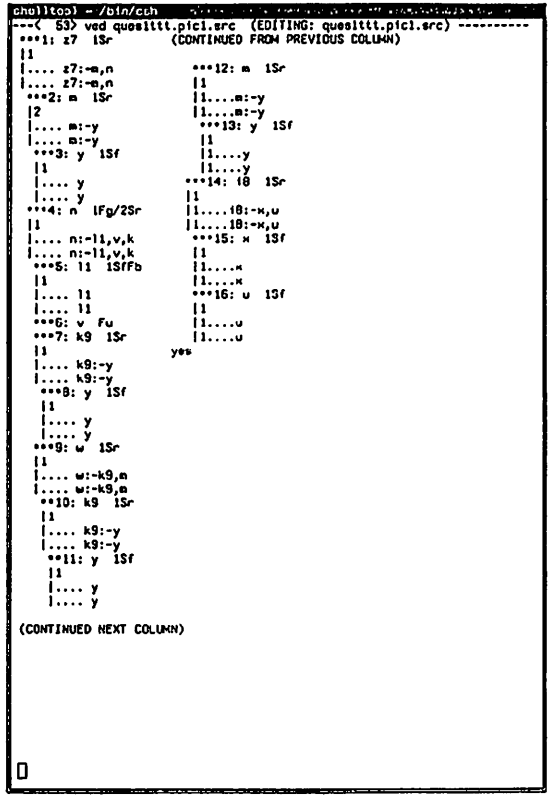

Figure 1: TTT Question 2

```
?- o.
** (1) Call : o?
** (2) Call : f?
** (3) Call : f1?
** (3) Exit : f1?
** (2) Exit : f?
** (4) Call : g?
** (5) Call : h?
** (5) Exit : h?
** (6) Call : prolog_error(UNDEFINED PREDICATE, [1])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE, [1])?
** (5) Redo : h?
** (5) Fail : h?
** (7) Call : j?
** (8) Call : f1?
** (8) Exit : f1?
** (7) Exit : j?
** (9) Call : k?
** (10) Call : j?
** (11) Call : f1?
** (11) Exit : f1?
** (10) Exit : j?
** (12) Call : f?
** (13) Call : f1?
** (13) Exit : f1?
** (12) Exit : f?
** (9) Exit : k?
** (14) Call : l?
** (15) Call : t?
** (15) Exit : t?
** (16) Call : f2?
** (16) Exit : f2?
** (14) Exit : l?
** (4) Exit : g?
** (1) Exit : o?
yes
?-
```

Figure 2: SPY Question 2

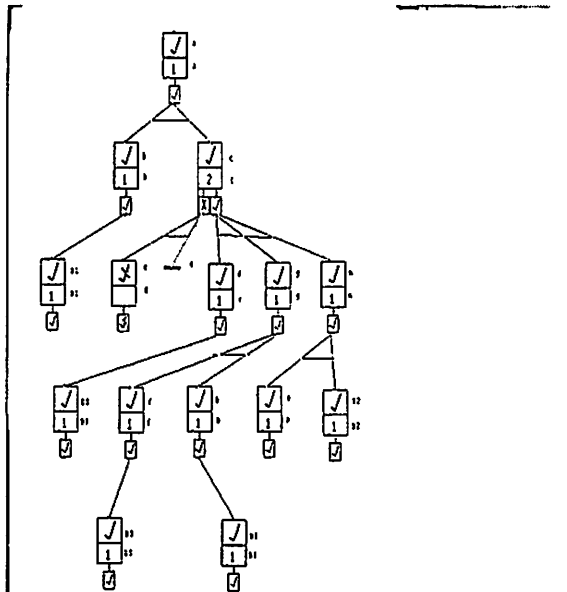

Figure 3: TPM* Question 2

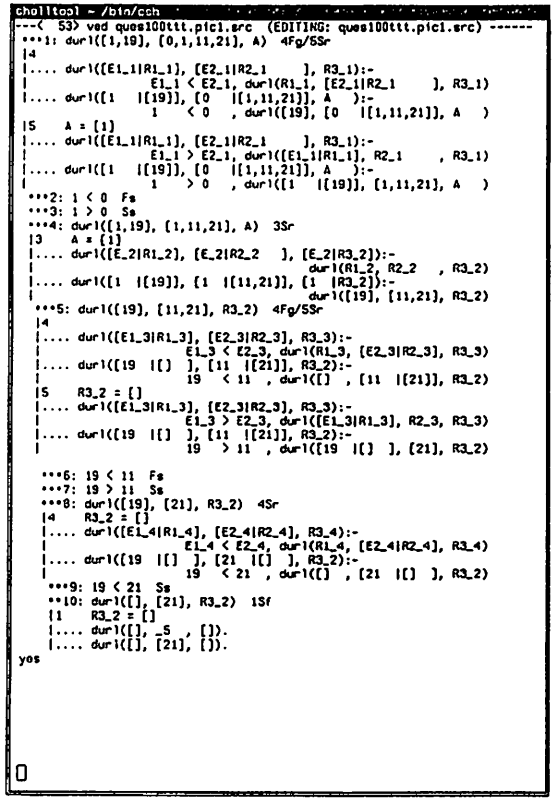

Figure 4: TTT Question 4

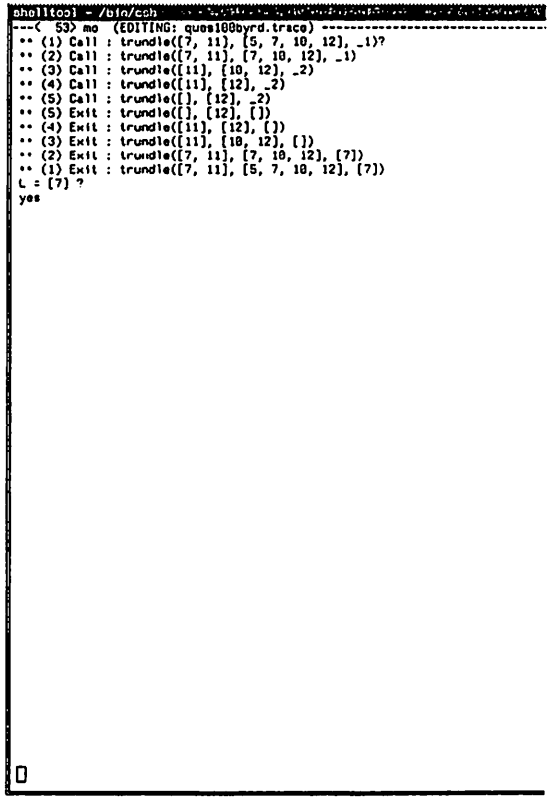

Figure 5: SPY Question 4

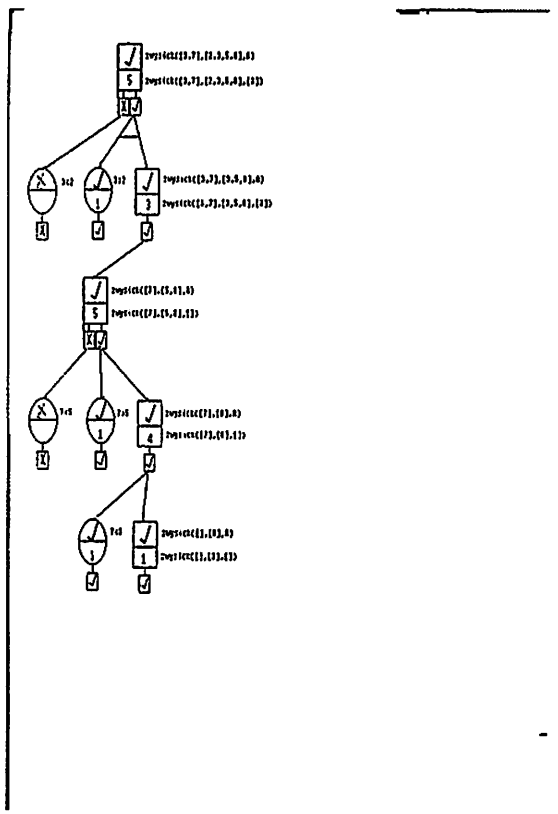

Figure 6: TPM* Question 4