*Int. J. Human–Computer Studies* (1997) **47,** 289–322

# Comparison of contrasting Prolog trace output formats

MUKESH J. PATEL
*Department of Psychology, University of Newcastle, Newcastle upon Tyne, UK*

BENEDICT DU BOULAY†
*School of Cognitive and Computing Sciences, The University of Sussex, Brighton, BN1 9QN, UK*

CHRIS TAYLOR
*Computer Science Department, City University, London, UK*

*(Received 19 March 1996 and accepted in revised form 10 February 1997)*

This paper reports on a comparative study of three Prolog trace packages. Forty-three students of an introductory Prolog course solved five different Prolog programming problems in each of three different conditions (using isomorphic problem variants to disguise recurring tasks). Each of the three conditions provided subjects with static screen-snapshot-mockups derived from one of three different trace packages ("conventional" Spy; "graphical AND/OR tree-based" TPM*; "informative textual" EPTB). When traces explicitly displayed the information asked for in the problem, subjects solved the problems more quickly. Conversely, when trace output obscured the required information (or necessitated difficult detective work to uncover the information), solution times were longer and answers less accurate. Deciding on a "good" format for display is thus a task-dependent decision, and impacts directly on the user's cognitive ability to solve a problem. © 1997 Academic Press Limited

## 1. Introduction

The ongoing development of visual programming languages and program visualization techniques is underpinned by the assumption that program design and debugging is made easier by diagrammatic or graphic representations (see Price, Baecker & Small, 1993, for a review). Theoretical accounts of how external representations work suggest that diagrammatic or graphical representations are not automatically more effective than informationally equivalent sentential or textual representations. A diagrammatic representation is likely to be more computationally efficient than a sentential representation if it reduces search among, aids recognition of or assists inference from information items in the external problem representation (Larkin & Simon, 1987; Larkin, 1989). The interplay between the internal cognitive processes of the problem-solver and the external representations on which they operate is still an active area of research (see Scaife & Rogers, 1996, for a critical review of this issue). Whether such efficiency gains are realized in practice depends on the nature of the problem-solving task and the degree of expertise and experience which the problem-solver can bring to bear on that task (see

† Corresponding author.

1071-5819/97/080289 + 34$25.00/0/hc970119

e.g. Petre & Green, 1990). In the case of programming language notations, the empirical evidence for the predicted value of graphic program language notations is not strong (see e.g. Green, Petre & Bellamy, 1991). By contrast, Cunniff and Taylor (1987) have shown that comprehension of graphically represented program segments was both faster and more accurate than that of textually represented equivalents though performance may have been affected by visual aptitude.

This paper is concerned with program trace output, an external representation typically used to assist in understanding and debugging the behaviour of a program, itself also typically available as an external representation. For an expert programmer the trace will often offer no information that could not, in principle, be inferred by scrutiny of the program itself.† In practical terms however, especially in debugging, the program trace plays a crucial role in that it enables a comparison between what the programmer believes a program should do and what it actually does. Reasoning about programming behaviour may not only be "forwards" or "backwards" (Green, 1977), but may be abductive, from the trace to the program (e.g. "which bit of the program produced this behaviour and why") or deductive, from the program to the trace (e.g. "where is the effect of this piece of code?"). The usefulness of the trace will depend on the programmer's declarative knowledge of the programming language and "strategic knowledge" of what to use the trace for and how to use it, together with processing constraints such as working memory capacity, visual discrimination capability and so on (Davies, 1993*b*).

Green and his colleagues have delineated many of the factors (such as "role expressiveness") which make a programming language notation psychologically effective (see e.g. Gilmore & Green, 1988; Green, 1989, 1991). Less effort has been expended on the effectiveness of representation of trace outputs from programs, but it is clear that they can be understood in similar terms and in terms of display-based reasoning.

The role of external representations in programming has been explored by Green, Bellamy and Parker (1987) in their "Parsing/Gnisrap" model of program development and by Davies (1993*a*) in his experiments on the tradeoff between the use of working memory and the display for storing fragments of an evolving program. Davies' experiments, in particular, suggest that one facet of programming expertise is precisely the ability to make good use of (and, indeed, to depend on) the display as an external representation. While his work was not concerned with trace outputs it indirectly underlines their importance because it elucidates how display-based strategies, in general, provide a means for coping with the complexity of programming tasks.

Two issues are explored here in relation to trace outputs for Prolog programs. The first concerns the relative efficiency of informationally equivalent textual and graphic trace notations in support of the kind of information lookup and inference tasks that occur in debugging simple programs. The second issue concerns the relationship between the trace notation and the program notation from which it derives. In a comparison of informationally equivalent textual and graphic traces, we might expect to observe the kinds of speedup in information access predicted by Larkin's models for a well-designed graphic trace notation. But we might also expect to see some counteracting slowdown for

† There are cases where this is obviously untrue, for example, debugging programs involving timing interactions or those affected by external events.

graphic trace notations where the debugging task required trace/program comparisons, if the *graphic* trace notation items were less easily matched than textual trace notation items to their corresponding *textual* Prolog program components.

## 1.1. PROLOG AND PROLOG TRACER OUTPUT FORMAT

In the study reported here we focus on the extent to which the format of Prolog tracer output helps to determine its usefulness to novice Prolog programmers. It is assumed that format is an important (though clearly not the sole) determiner of the clarity of information in programming notations (Green, 1991). For a programming language as internally complex as Prolog this is especially the case because the format of the tracer output can be a significant determiner, amongst other things, of both the overall perspective and the level of detail about program execution. Both these aspects, perspective and the level of detail, accordingly affect the ease of *access* to the information given by a particular combination of a tracer and its format (Patel, du Boulay & Taylor, 1991*a*, *b*). Other significant determiners of information access are the control interface to the tracer and the user herself; no matter how clear or concise the information presented is, if it fails to match the user's cognitive model of the program domain and the language it is unlikely to be optimally useful. By "control interface to the tracer" we mean the manner in which the user may control the tracer itself—choosing which predicates to examine, how to set breakpoints, methods of controlling the amount and level of information both globally and at different points in the execution, ways of moving around the trace output and so on. What might otherwise be an excellent format could be marred by poor user control and what otherwise might be regarded as an impoverished format may continue to be effective where control issues work well. For example, Davies' manipulation of the control interface to an editor has indicated how important this aspect of a tool is in increasing or reducing programming errors (Davies, 1993*a*).

This study concentrates on format rather than on control or user characteristics. Our research aim was *not* to find out why some novices have problems with learning Prolog but to evaluate the relative usefulness of certain trace outputs in solving simple problems. While a better understanding of the ways in which tracers as a whole are useful (rather than simply their static outputs) would no doubt lead to a better cognitive model of Prolog novices' problems, this too was beyond the scope of this work.

Prolog is a complex and powerful programming language. As with all programming languages many key aspects remain *implicit* (or "hidden") during program execution, which adds to the difficulty in comprehension and confusion among novice programmers. Prolog is special in that the internal mechanisms governing flow of control and variable binding are more complex than many procedural languages (such as Basic or Pascal) for which there is ample evidence of novice confusion about flow of control and variables (see e.g. du Boulay, 1986, for a brief review). Prolog on-line help systems can be typically divided into a number of groups which include both high-level debugging and tutoring systems and monitoring systems or *tracers*.

Tracers, in general, are used by novices for a variety of tasks such as learning the language as well as program development and debugging (see e.g. Mann, Linn & Clancy, 1994). In the case of Prolog, trace output can enhance a novice's understanding of the implicit aspects of Prolog execution such as flow of control, backtracking, variable binding and variable unbinding and the general relationship between the static code of the program and the dynamic behaviour it engenders via a query. Of course, tracers do not provide information on how to write Prolog programs to achieve specific goals nor do they provide guidance and reasons for a program's failure, though they are very helpful in debugging tasks. Some of these areas are covered by more sophisticated debugging systems (see e.g. Brna, Brayshaw, Bundy, Dodd, Elsom-Cook & Fung, 1991) or tutoring systems (see e.g. Looi, 1991; Gegg-Harrison, 1991) which were not evaluated in this study. For a more extended account of criteria for evaluating Prolog tracers, see Rajan (1991) and Taylor, du Boulay and Patel (1991).

The hidden mechanisms of Prolog can be described and/or explained in more than one way (Pain & Bundy, 1987) with different *perspectives* emphasising different aspects, such as variable binding, flow of control, recursion, search space, etc. 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 languages; emphasis on one aspect, such as flow of control, often precludes the possibility of other aspects without loss of clarity.

Format is another major determinant of clarity of information. For example, a perspective focusing on the flow of control in program execution can be presented *inter alia* as a graphical AND/OR tree rooted at the top and growing downwards (as in TPM Eisenstadt & Brayshaw, 1988) or as a left-to-right, sideways, textual tree (as in Mellish, 1984). Independent of the perspective, tracers provide differing amounts of information necessary for reconstructing the whole "story" of a program's execution, though in some sense each tracer is a more or less equally valid (and theoretically adequate) description. In some tracers the user has to make more complex inferences about what has happened than in others.

Apart from perspective and format, tracers vary in terms of information content. As with perspective, up to a certain extent, information about how a program is executed does not vary across trace outputs (even if perspective and format do). The formal properties of Prolog leave no room for ambiguity in execution; so, given an adequate grasp of the underlying logic and the source code, it is usually possible to comprehend a trace output.† What does vary, however, is the level of detail and explicitness of information. Together, these affect the ease of *access* to information, and it is this property of trace outputs that we regard as an important determiner of tracer usefulness. Usually, it is far more appropriate to compare tracers in terms of the extent to which information about key aspects of Prolog is more or less explicit rather than whether it is absent or present. Of course, the ease of access to information will also be partly determined by format. We will return to this issue in the next section.

## 2. Details of tracer outputs

Three tracer output formats were compared: Spy (or Byrd Box) (Byrd, 1980), Enhanced Prolog Tracer for Beginners (EPTB) (Dichev & du Boulay, 1989) and a variant

† There are minor differences in the way that different versions of Prolog implement, e.g. clause matching. Some systematically scan through all clauses with a given functor and the correct arity, others employ various pre-matching techniques. These differences ought to show up in any trace output that worked at that level of detail.

on the Transparent Prolog Machine (TPM) (see e.g. Eisenstadt, Brayshaw & Paine, 1991).

The tracers were not used truly as tracers in a dynamic way. In each case an appropriate screen dump of the relevant tracer 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. As we have already indicated, our intention was not to examine the tracers "in the round" in the manner of Mulholland (1994, 1995) but to focus on the influence of format and level of detail in *static* trace outputs.

### 2.1. SPY TRACE OUTPUT

Spy is a basic, linear† textual tool and is included in this study because subjects were familiar with it and in order to provide a baseline for comparing with others. The version used in this study (POPLOG Prolog) did not show system goals and did not emphasize trace structure via indentation. 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 EPTB. Given the basic lack of clarity in information presentation, Spy was not expected to perform better than TPM* and EPTB. Figure 1 shows a simple program (top) and query together with its Spy trace (top, right).

### 2.2. EPTB TRACE OUTPUT

EPTB (Enhanced Prolog Tracer for Beginners) is a linear, textual tracer developed by Dichev and du Boulay (1989). Its most characteristic feature is an emphasis on providing information about data structures and variable binding in the trace output. It also has some other features not present in a Spy tracer, such as a facility for distinguishing between different reasons for failure of goals plus a wide range of options for adjusting the degree of detail shown. The tracer generally makes use of labels and symbols to describe different sections of a trace output (like PTP, see Eisenstadt, 1984). Hence, it is expected to be more helpful than Spy for solving problems involving data structures and variable binding. The tracer can be set to include or exclude messages about various aspects of Prolog execution history. Figure 1 (bottom left) shows the EPTB trace for the same program and query, with the trace options set in one particular way.

### 2.3. TPM* TRACE OUTPUT

The TPM (Transparent Prolog Machine) is a tracing and tutoring tool which makes use of a modified and extended AND/OR tree representation, known as the "AORTA" representation. TPM exists in a number of forms: As a notation used in courses offered by the Open University (e.g. Eisenstadt & Dixon, 1988) and as described in the research

† A linear tracer always adds new trace statements to the growing *end* of the trace and does not either overwrite or insert material elsewhere within it.

```
s(b).
r(b).

p(X):- q(X), r(X).
p(c).

q(a).
q(X):- s(X).
```

```
?- p(Y).
** (1) Call : p(_1)?
** (2) Call : q(_1)?
** (2) Exit : q(a)?
** (3) Call : r(a)?
** (3) Fail : r(a)?
** (2) Redo : q(a)?
** (4) Call : s(_1)?
** (4) Exit : s(b)?
** (2) Exit : q(b)?
** (5) Call : r(b)?
** (5) Exit : r(b)?
** (1) Exit : p(b)?
Y = b ?
yes
```

```
?- p(Y).
** (1) Call : p(Y)
1) p(X_1):-q(X_1),r(X_1)
*> p(Y):-q(Y),r(Y)
-
** (2) Call : q(Y)
1) q(a)
*> q(a)
** (2) Exit : q(a)
-
** (3) Call : r(a)
** (3) Fail-match : r(a)
** (2) Redo : q(a)
** (2) Retry : q(Y)
2) q(X_2):-s(X_2)
*> q(Y):-s(Y)
-
** (3) Call : s(Y)
1) s(b)
*> s(b)
** (3) Exit : s(b)
** (2) Exit : q(b)
-
** (4) Call : r(b)
1) r(b)
*> r(b)
** (4) Exit : r(b)
** (1) Exit : p(b)
Y =  b ??
yes
```

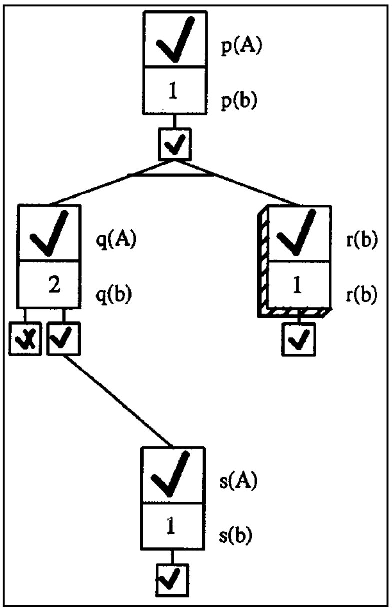


FIGURE 1. Simple program (top left) with Spy trace (top right), EPTB trace (bottom left) and TPM* trace (bottom right).

literature (see e.g. Eisenstadt & Brayshaw, 1988); and as implementations available on various machines in differing degrees of conformance with the printed versions. At the time of the experiment we had access to a commercially available version of the TPM, marketed by Chemical Design Limited and hence referred to as CDL-TPM. We used the "detailed view" tree notation from this tracer as the graphical notation for the experiment. However, in this detailed notation, the CDL-TPM tracer could only display at once a single node plus its parent and immediate children. Thus, the tracer itself could not generate "detailed view" trees that were big enough for the experimental examples, which involved up to 20 nodes or so. Consequently, the static traces used were constructed artificially, using the same notation, but applied to bigger trees than the tracer could actually show in that notation.† To emphasize the fact that the "traces" were not actually generated from the CDL-TPM tracer, the tree notation used is referred to as "TPM*".

The spatial layout of TPM* provides a clearer and less cluttered view of the structure of the search space than the two linear textual tracers. On the other hand, the use of a vertical tree format limits the horizontal space available between sibling nodes for displaying textual information such as predicate names, arguments and variable bindings. (By contrast, in linear, unindented textual tracers, virtually the entire window width is available for displaying such information.) This point is not relevant to the simple examples in the experimental study, in which the call terms were all fairly short syntactically. However, in the CDL-TPM tracer, it rapidly becomes a problem when long predicate names or long terms (e.g. lists) are used, necessitating the use of expandable and scrollable sub-windows for textual information, with a resultant decrease in the efficiency of access to textual information. (*Note*: In other versions of the TPM, a simultaneous textual trace using a Spy-like notation is available in a separate sub-window—although this takes up further screen space.) A tracer output in TPM* for the same program above is shown in Figure 1 (bottom right).

## 3. Trace outputs: information and format

All three tracers have slightly different perspectives, though this is not assumed to have any significant effect on their usefulness for our experimental task. For the purpose of this study it was assumed that the three tracers provide the minimal—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.

### 3.1. INFORMATION ACCESS

Apart from the effect of perspective, how do trace outputs vary in terms of overall information content? In this context, the term *information* is used in a specific way; it

† With the wisdom of hindsight, it might well have been better to have used detailed, printed trace diagrams from the TPM literature.

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 access to this information, but they can vary in terms of the exact nature of information provided. Part of the variation can be due to the level of detail. For example, the Spy trace does not indicate which clause of a predicate is being used at any point, whereas EPTB and TPM* do. Spy refers to program variables by their internal names such as "_405", while CDL-TPM systematically labels variables with letters (current implementations of TPM and printed TPM diagrams make use of the programmers' variable names, suitably subscripted), and unlike both, EPTB 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.

Further, trace outputs have different degrees of explicit information. For example, information about the number of sub-goals of a clause can be highly implicit, as in a Spy trace, or fairly explicit, as in TPM* (as long as the clause succeeds) and in EPTB (independent of whether the clause succeeds or not). While it is not difficult to provide examples to refine our notion of level of detail and explicitness of information, in reality these two aspects are often closely inter-related. However, this is not a serious drawback as long as it is clear that whatever the level of detail and the degree of information explicitness, it is the effect on access that determines a trace output's usefulness. Thus, it follows that simply having more detailed or explicit information does not necessarily increase usefulness, because too much detail may be a hindrance in some cases. This tension between being explicit and overwhelming the user with "unnecessary" detail (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.

We are taking a rather "all or nothing" view here because of the nature of the experiment, which was based on static snapshots of trace outputs. In employing a tracer for a realistic task the user would expect to be able to adjust the level of detail dynamically both globally and locally. Note that with real tracers solution times might well have been *longer*.

### 3.2. FORMAT: GRAPHIC AND TEXTUAL

Aesthetic aspects of graphic representation can improve communication efficiency (Shu, 1988), but such improvement often depends on the nature of the information being displayed. In the case of Prolog programs there is a limited number of ways in which information about flow of control and backtracking can be graphically displayed. For example, flow of control can be represented as an AND/OR tree or an OR-tree (Hook, Taylor & du Boulay, 1990) or in terms of flow of satisfaction arrows (Clocksin & Mellish, 1981). In such cases, the spatial arrangement of certain aspects of information plays a crucial role in capturing essential and important relationships and would be expected to be differentially helpful in solving problems related to flow of control and backtracking. So assuming that the information content of a tracer is fixed for the most significant aspects of Prolog, how can format effect their usefulness?

Trace outputs can be displayed in mainly graphic or mainly textual format. This basic distinction will be described before a detailed consideration of their effect on trace output usefulness. The following list of advantages of non-linear graphic format also serves to highlight the disadvantages of linear text format. It is included to anchor some of the basic differences between non-linear graphic and linear text formats. It is, of course, possible to have non-linear textual formats—which have some of the advantages of both (Taylor *et al*., 1991; Taylor, du Boulay & Patel, 1994).

1. Correlation between spatial display and information, e.g. relative distance and proportion or bringing together related information items.
2. Clarity in display of non-linearly ordered information such as loops and backtracking.
3. Enables zooming in and out to access the essential relationship between information points.
4. Can display more than one perspective, if information points are related across more than one dimension. Simultaneous display of different perspectives is not necessary.
5. Greater possibility of displaying dynamic processes (animation) with real time updates. Though possible in text formats they may lack impact and clarity.

Note that some of the foregoing advantages are not exclusive to graphic formats (since non-linear textual formats share some of the same features), and others are of limited value unless augmented with textual information. The format of trace outputs can have an important effect on the extent to which different aspects of Prolog can be presented clearly and simultaneously. Its interaction with perspective and effect on ease of access to information has already been described. For example, TPM*'s graphic format supports a clearer presentation of flow of control information which renders temporal (historical) information about clause matching a bit less accessible; broadly speaking, the opposite is the case for EPTB.

Aside from trivial cases, the distinction between graphic and textual format is rarely clear cut. Even in a standard, linear textual format the serial ordering of information about states at each step in a program run reflects an implicit notion of temporal ordering; information at any step $n$ is partly dependent on information at step $n-1$ and will itself determine the information content of step $n+1$. Serial ordering of information in such a case is not textual but *graphical*. Conversely, a graphical tracer would be of limited value without embedded textual information such as the values of bound variables. However, a crude distinction between the two categories of tracer formats is sufficient here because our aim is to concentrate on differences in usefulness due to very broad parameters; e.g. the difference due to the representation of flow control as a graphical AND/OR tree or as a linear ordering of textual information.

The relative advantages of graphic formats over linear textual formats might intuitively appear to be considerable. However, the major advantages of a graphic format may be adversely affected by the *modality mismatch* between textual program code (assuming traditional textual Prolog) and its graphic trace. The difference in modalities between

Prolog source code and its representation in a graphic trace is a likely source of ambiguity, which may influence ease of access to information. Thus, the advantages of graphic format cannot be assumed to be uniformly beneficial in all applications.

To recap, assuming that all the traces could be used to obtain the information required (in some cases in combination with a display of the corresponding source code and a basic knowledge of Prolog execution), but that they varied in terms of access to the information, how would this variation in access affect the subject's ability to obtain the information? 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 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 the less time spent on solving the problem and the less likelihood of errors in the solution.

## 4. Task and motivation

The trace output format evaluation task was primarily designed to check for the effect of format on the relative usefulness of traces in solving simple Prolog problems. The task was presented on a workstation and response time and errors were collected. Each subject was assumed to know about the underlying principle of Spy traces (e.g. Byrd Box) and was given a tutorial on the other two trace outputs.

The task was a very simplified and abstract version of part of a normal debugging task. The subject group, while adequately proficient in Prolog, had to learn about two new trace outputs (TPM* and EPTB) in order to do the task. Because of this learning load, it was felt that subjects should not have to learn to manipulate actual tracers. Thus the problem solving task was not interactive. Subjects were given a static copy of the final trace output of the relevant program to enable them to solve the problems. Solution to the problems did not depend on being able to run the programs, which were very simple as the examples show. A subject was presented with a problem and a single window of trace output and required to solve the problem with aid of that trace. Hence, it was a static version of a normal debugging task but adequate to check for the impact of format in solving one sort of Prolog problem.

Since there are information content differences between the trace outputs, in general, the stimuli problems were designed to evaluate access to information which was common to all three trace types. Solutions to problems did not depend on obvious major differences such as the "cut", which can be very explicitly and graphically displayed in TPM* but is totally obscure in Spy. However, there are minor differences between the tracers and the purpose of this paper is to investigate the problem $\times$ tracer interaction.

The problems were limited by the non-interactive nature of the task and by the requirement that they should be concerned largely with accessing information from the tracers as opposed to focusing on obscure features of the problem domain. That is, they were concerned with activities that form *part of* the debugging process but were not supposed to be tests of the subjects' debugging ability.

A more sophisticated study than this might have extracted the problems from actual course materials in order to ensure their ecological validity. This was not done, but we tried to ensure that the problems chosen were representative of the kinds of sub-tasks that might occur when novice programmers are trying to reconcile a program, its behaviour and a trace. We ensured as far as possible that solutions depended on basic understanding of Prolog rather than high levels of expertise.

The null hypothesis was that it would be possible to answer each question with the help of each trace output without significant differences in either response time or mean error.

## 4.1. PROBLEMS

The stimuli consisted of five problems. Three were very simple problems which could be solved with information on backtracking and clauses tried. The solutions to the other two problems depended on information about recursion, system goals, goals with variables and list manipulation. Each problem was presented three times in overall random order; once with each trace output. Care was taken to disguise any similarity between question across different trace output types. This was achieved by altering words and phrases in the problem statements, as well as predicate and variable names. For examples of "disguised" EPTB and TPM* questions, see the appendix.†

Each of the five problems used in described and illustrated in this section in their Spy trace form, together with the corresponding traces for EPTB and TPM*. Note that, because of the disguising, the predicate and variable names in the corresponding EPTB and TPM* traces are not the same. Each of the problems was presented with a multiple choice question the order of whose answers was randomised on each presentation of the question.

Problem 1: To solve this simple problem, subjects have to work out how often particular procedures are called. To ensure that subjects used the trace output rather than work from the program, it is presented without the program, see Figure 2.

The **Spy** textual trace largely requires linear scanning and counting. The first option requires that the subject establishes that there is only a single line of the form ** `(nn) Call: t`, where "nn" means a small integer. The second option requires that the subject count-up lines of the form ** `(nn) Call: f1`. The third option requires the subject to establish that there are no lines of the form ** `(nn) Call: h`. The fourth option requires that the subject find two lines of the form ** `(nn) Exit: g`.

The **EPTB** textual trace also required linear scanning and counting, see Figure 3. While there are more lines to scan, the task is essentially similar to the Spy version.

For the **TPM*** trace the subjects have to count the number of call boxes labelled with the name of the procedure in question, see Figure 4. Checking the success of a call requires noticing that its clause status box shows a tick.

Problem 2: This problem, also presented without the program, requires subjects to find out how many subgoals a particular clause contained, see Figure 5. The same traces as in problem 1 are used.

† It is possible that variations in the wording of the questions or small changes in the names of variables and predicates may have contributed to reducing the equivalence of the questions.

```
Given the trace to be shown, which one of the
following statements is CORRECT?

1. There is one call of t*
2. f1 is called thrice
3. There are no calls of h
4. Two calls of g succeed
```

```
?- e.
** (1) Call : e
** (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, [i])
** (6) Fail : prolog_error(UNDEFINED PREDICATE, [i])
** (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 : e
yes
```

FIGURE 2. Spy Question 1 (top) and trace (bottom). The starred answer is correct.

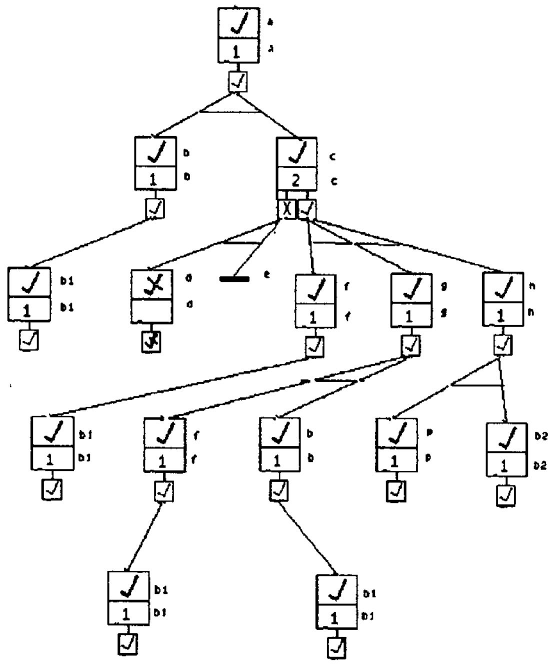
```
??-  k.
** (1) Call : k
1) k:-l,m
*> k:-l,m
-
** (2) Call : l
1) l:-l1
*> l:-l1
-
** (3) Call : l1
1) l1
*> l1
** (3) Exit : l1
** (2) Exit : l
-
** (4) Call : m
1) m:-n,o,a
*> m:-n,o,a
-
** (5) Call : n
1) n
*> n
** (5) Exit : n
-
** (6) Call : o

** (6) Fail-match : o
** (5) Redo : n

** (5) Fail : n

** (4) Retry : m
2) m:-p,q,r
*> m:-p,q,r
-
** (5) Call : p
1) p:-l1
*> p:-l1
-
** (6) Call : l1
1) l1
*> l1
** (6) Exit : l1
** (5) Exit : p
-
** (7) Call : q
1) q:-p,l
*> q:-p,l
-
** (8) Call : p
1) p:-l1
*> p:-l1
-
** (9) Call : l1
1) l1
*> l1
** (9) Exit : l1
** (8) Exit : p
-

(CONTINUED NEXT COLUMN)
```

```
(CONTINUED FROM PREVIOUS COLUMN)

** (10) Call : l
1) l:-l1
*> l:-l1
-
** (11) Call : l1
1) l1
*> l1
** (11) Exit : l1
** (10) Exit : l
** (7) Exit : q
-
** (12) Call : r
1) r:-z,l2
*> r:-z,l2
-
** (13) Call : z
1) z
*> z
** (13) Exit : z
-
** (14) Call : l2
1) l2
*> l2
** (14) Exit : l2
** (12) Exit : r
** (4) Exit : m
** (1) Exit : k

yes
```

Key to Spy question 1

| Spy | | EPTB |
|---|---|---|
| e | = | k |
| f | = | l |
| f1 | = | l1 |
| g | = | m |
| h | = | n |
| i | = | o |
| k | = | q |
| t | = | z |

FIGURE 3. EPTB Question 1 trace.



Key to Spy question 1

| Spy | | TPM* |
|---|---|---|
| e | = | a |
| f | = | b |
| f1 | = | b1 |
| g | = | c |
| h | = | d |
| i | = | e |
| k | = | g |
| t | = | p |

FIGURE 4. TPM* Question 1 trace.

```
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?

1. The first clause for k has two subgoals*
2. The first clause for g has five subgoals
3. The second clause for g has five subgoals
4. e is defined by a single recursive clause
```

FIGURE 5. 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 FALSE?

1. The first clause for g may have more than two subgoals
2. The second clause for g has exactly three subgoals
3. the second clause for f is not called
4. i succeeds*
```

FIGURE 6. Spy Question 3.

In the **Spy** trace output for the first three options this information is not explicitly stated and needs to be inferred. Because Spy does not assign numbers to matching clauses during execution, this simple problem is rendered more complicated. Subjects have to infer that the first "call" line for a procedure corresponds to the matching of the first clause of that procedure and locate a corresponding "exit" line further down. All calls to the sub-goals appear between these two lines, so subjects have to count all sub-goal "call" and "exit" line pairs at the relevant level. While doing so they have to ensure that calls to sub-sub-goals of the relevant sub-goals which also appear in the same intervening section are not included in the count. The fourth option asks the subject to ascertain whether "e" is defined by a single recursive clause. Given that the goal "e" succeeds, there would have to be an "e" sub-goal for this to be true.

With an **EPTB** trace output the subject can locate a line showing that clause in the trace and count off the number of sub-goals shown on that line. This information is explicitly displayed. The more long-winded counting strategy, described for Spy, can also be applied.

In **TPM*** the required information is available explicitly in the trace and subjects need only count the number of sub-goal nodes that are child-nodes of a box for the relevant clause. In the case of the fourth option, one of the child nodes of "e" (in fact "a" in the TPM* trace) would also have to be "e".

Problem 3: For this problem subjects are presented with the same traces as Problems 1 and 2, but have to choose a *false* statement from a choice of four. The required false answer (the fourth option) is that the call to "i" succeeds, where, in fact, the call fails because no procedure with the corresponding name and arity has been defined; the Prolog program was not given, see Figure 6.

```
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"?

1. Once
2. Three times*
3. Four times
4. Not at all
```

```
?-   trundle([7,11], [5,7,10,12], L).
** (1) Call : trundle([7, 11], [5, 7, 10, 12], _1)?
** (2) Call : trundle([7, 11], [7, 10, 12], _1)?
** (3) Call : trundle([11], [10, 12], _2)?
** (4) Call : trundle([11], [12], _2)?
** (5) Call : trundle([], [12], _2)?
** (5) Exit : trundle([], [12], [])?
** (4) Exit : trundle([11], [12], [])?
** (3) Exit : trundle([11], [10, 12], [])?
** (2) Exit : trundle([7, 11], [7, 10, 12], [7])?
** (1) Exit : trundle([7, 11], [5, 7, 10, 12], [7])?
L = [7] ?
yes
```

FIGURE 7. Spy Question 4 (top) and trace (bottom).

For the **Spy** trace the subject's procedure should be largely the same as in Problem 2, except for the correct/false inversion. The first two options are establishing the number of sub-goals in a clause. In the third option the subject need to check whether there is a "`fail`" at the immediate subsidiary level between a ** `(nn) Call:f` and

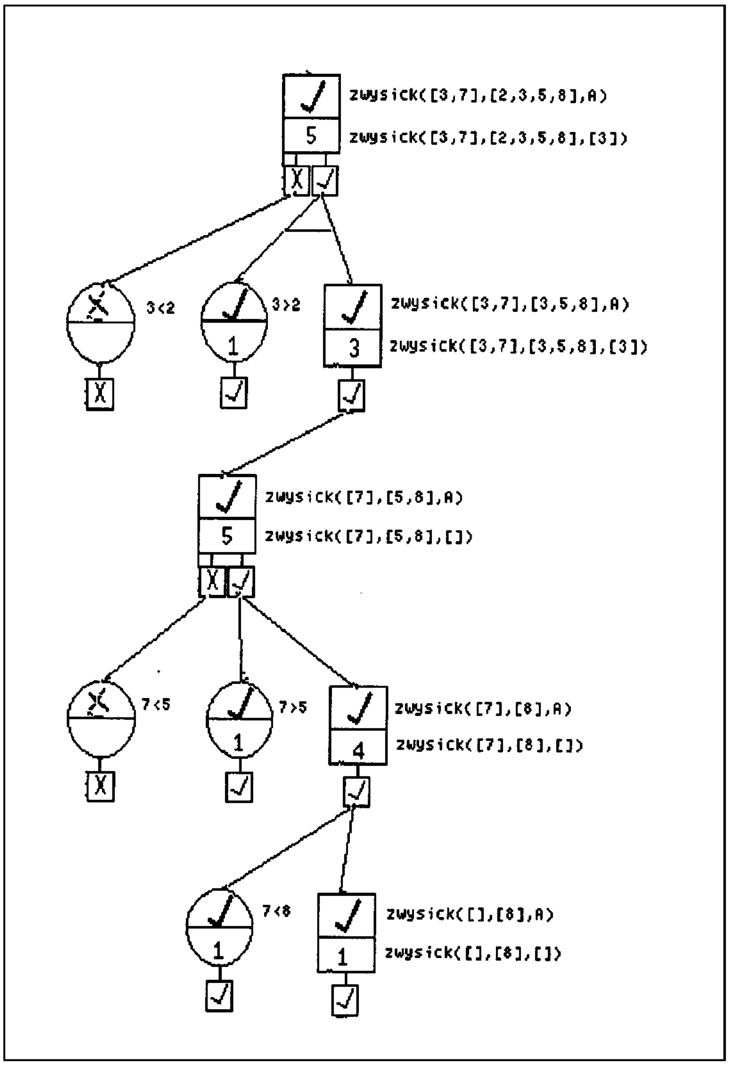
```
??-  mayrphi([9,19], [1,9,11,30], L).
** (1) Call : mayrphi([9,19],[1,9,11,30],L)
4) mayrphi([H1_1|T1_1],[H2_1|T2_1     ],Others_1):-
                 H1_1<H2_1,mayrphi(T1_1,[H2_1|T2_1     ],Others_1)
*> mayrphi([9   |[19]],[1   |[9,11,30]],L        ):-
                 9   <1   ,mayrphi([19],[1   |[9,11,30]],L        )

** (1) Retry : mayrphi([9,19],[1,9,11,30],L)
5) mayrphi([H1_1|T1_1],[H2_1|T2_1     ],Others_1):-
                 H1_1>H2_1,mayrphi([H1_1|T1_1],T2_1      ,Others_1)
*> mayrphi([9   |[19]],[1   |[9,11,30]],L        ):-
                 9   >1   ,mayrphi([9   |[19]],[9,11,30],L        )
-
** (2) Call : mayrphi([9,19],[9,11,30],L)
3) mayrphi([H1_2|T1_2],[H1_2|T2_2   ],[H1_2|Others_2]):-
                                  mayrphi(T1_2,T2_2    ,Others_2)
*> mayrphi([9   |[19]],[9   |[11,30]],[9   |Others_2]):-
                                  mayrphi([19],[11,30],Others_2)
-
** (3) Call : mayrphi([19],[11,30],Others_2)
4) mayrphi([H1_3|T1_3],[H2_2|T2_3],Others_3):-
                     H1_3<H2_2,mayrphi(T1_3,[H2_2|T2_3],Others_3)
*> mayrphi([19  |[]  ],[11  |[30]],Others_2):-
                     19  <11  ,mayrphi([]  ,[11  |[30]],Others_2)
so:  L =>  [9 | Others_2]

so:  L =>  [9 | Others_2]
** (3) Retry : mayrphi([19],[11,30],Others_2)
5) mayrphi([H1_3|T1_3],[H2_2|T2_3],Others_3):-
                     H1_3>H2_2,mayrphi([H1_3|T1_3],T2_3,Others_3)
*> mayrphi([19  |[]  ],[11  |[30]],Others_2):-
                     19  >11  ,mayrphi([19  |[]  ],[30],Others_2)
so:  L =>  [9 | Others_2]
-
** (4) Call : mayrphi([19],[30],Others_2)
4) mayrphi([H1_4|T1_4],[H2_3|T2_4],Others_4):-
                     H1_4<H2_3,mayrphi(T1_4,[H2_3|T2_4],Others_4)
*> mayrphi([19  |[]  ],[30  |[]  ],Others_2):-
                     19  <30  ,mayrphi([]  ,[30  |[]  ],Others_2)
so:  L =>  [9 | Others_2]
-
** (5) Call : mayrphi([],[30],Others_2)
1) mayrphi([],_1  ,[])
*> mayrphi([],[30],[])
so:  L =>  [9]
** (5) Exit : mayrphi([],[30],[])
** (4) Exit : mayrphi([19],[30],[])
** (3) Exit : mayrphi([19],[11,30],[])
** (2) Exit : mayrphi([9,19],[9,11,30],[9])
** (1) Exit : mayrphi([9,19],[1,9,11,30],[9])

L =  [9] ??

yes
```

Key to Spy question 4

| Spy | | EPTB |
|---|---|---|
| trundle | = | mayrphi |
| 7 | = | 9 |
| 11 | = | 19 |
| 5 | = | 1 |
| 10 | = | 11 |
| 12 | = | 30 |
| X | = | H1 |
| Xs | = | T1 |
| Y | = | H2 |
| Ys | = | T2 |
| Zs | = | Others |

FIGURE 8. EPTB Question 4 trace.

its corresponding ** `(nn) Exit:f` or ** `(nn) fail:f`. In the case of the fourth option, once the relevant subsection of the trace has been identified, the subject needs only to find the phrase "UNDEFINED PREDICATE" in order to confirm the statement.



**Key to Spy question 4**

| Spy | | TPM* |
|---|---|---|
| trundle | = | zwysick |
| 7 | = | 3 |
| 11 | = | 7 |
| 5 | = | 2 |
| 10 | = | 5 |
| 12 | = | 8 |
| L | = | A |

FIGURE 9. TPM* Question 4 trace.

As for Problem 2, the first two options in **EPTB**'s textual format require either systematic linear scanning, or looking up the information which is explicitly stated in the trace. For the third option, the subject can apply the same procedure as for the Spy trace or look for an explicit ** `(nn) Retry : f` line (in fact "l"). In order to solve the fourth option, the subject has to locate the relevant procedure call for "i" (in fact "o") and check its success or failure to match on the following line.

With the **TPM*** trace, the first three options are dealt with by counting within the groupings of sub-goals of the relevant clause box. In order to verify the statement in the

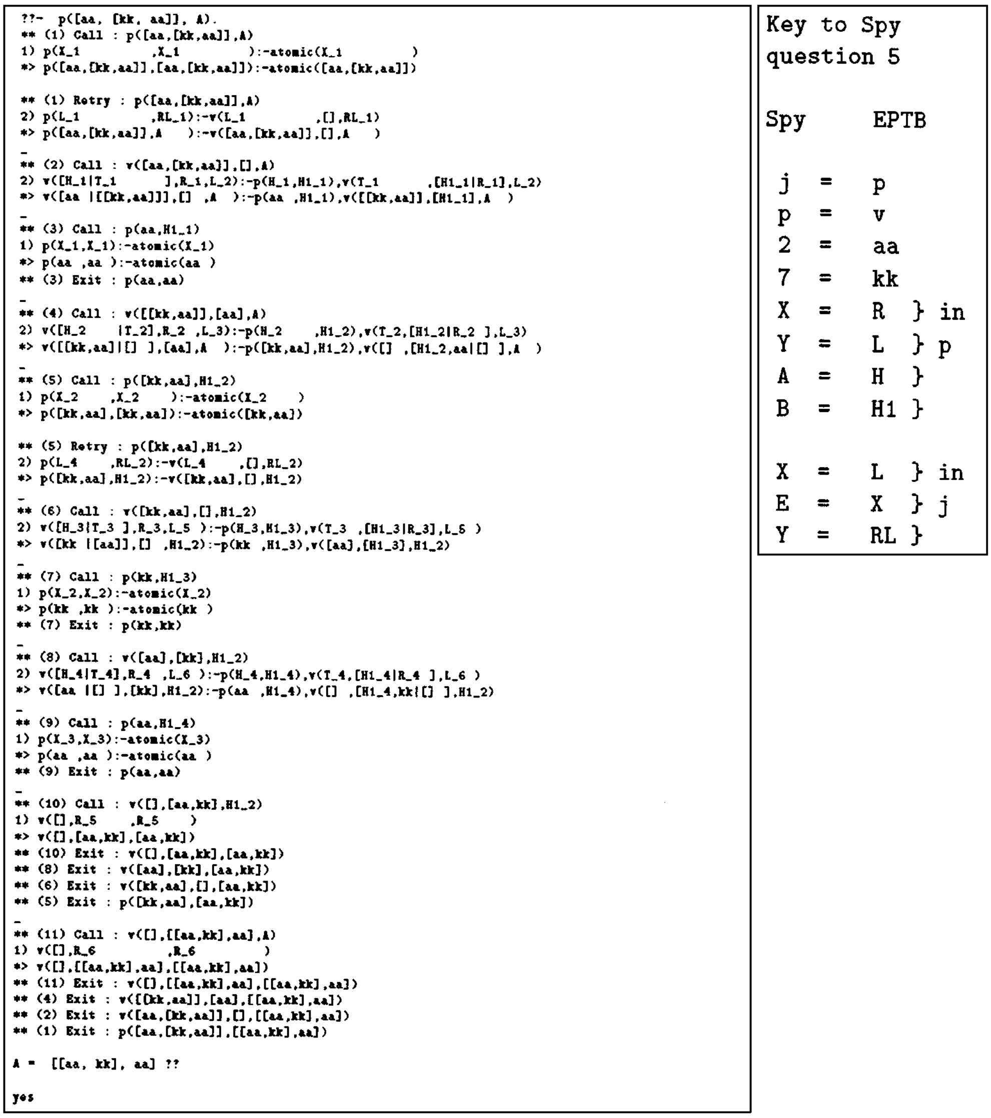
```
Suppose the goal

?- j([2, [7, 2]], X).

is evaluated against the program below:

p([], X, X).
p([A|R], X, Y):- j(A, B), p(R, [B|X], Y).

j(E,E):- atomic(E).
j(X, Y):- p(X, [], Y).

You are shown a number of statements, only one
of which is correct. With the help of the
trace you are shown, pick the CORRECT statement.

(N.B. A clause is "invoked" if its head matches
 a call - it will then succeed if all its
 subgoals succeed, and fail otherwise.)

1. 2nd clause of p is invoked 4 times*
2. 1st clause of p is invoked 3 times
3. 2nd clause of j is invoked 3 times
4. 1st clause of j is invoked 3 times
```

```
?- j([2, [7, 2]], X).
** (1) Call : j([2, [7, 2]], _1)
** (2) Call : p([2, [7, 2]], [], _1)
** (3) Call : j(2, _2)
** (3) Exit : j(2, 2)
** (4) Call : p([[7, 2]], [2], _1)
** (5) Call : j([7, 2], _3)
** (6) Call : p([7, 2], [], _3)
** (7) Call : j(7, _4)
** (7) Exit : j(7, 7)
** (8) Call : p([2], [7], _3)
** (9) Call : j(2, _5)
** (9) Exit : j(2, 2)
** (10) Call : p([], [2, 7], _3)
** (10) Exit : p([], [2, 7], [2, 7])
** (8) Exit : p([2], [7], [2, 7])
** (6) Exit : p([7, 2], [], [2, 7])
** (5) Exit : j([7, 2], [2, 7])
** (11) Call : p([], [[2, 7], 2], _1)
** (11) Exit : p([], [[2, 7], 2], [[2, 7], 2])
** (4) Exit : p([[7, 2]], [2], [[2, 7], 2])
** (2) Exit : p([2, [7, 2]], [], [[2, 7], 2])
** (1) Exit : j([2, [7, 2]], [[2, 7], 2])
X = [[2, 7], 2] ?
yes
```

FIGURE 10. Spy Question 5 (top) and trace (bottom).

fourth option subjects have to either identify a box with the relevant successful call to disprove the statement or seek out a horizontal bar which indicates that no clauses matching the call exist.

Problem 4: To solve this problem, which includes the program source code, subjects are asked about the number of times a particular clause is invoked when the program is executed to prove a given query. The correct answer is three times, see Figure 7.



```
??-  p([aa, [kk, aa]], A).
** (1) Call : p([aa,[kk,aa]],A)
1) p(X_1            ,X_1          ):-atomic(X_1          )
*> p([aa,[kk,aa]],[aa,[kk,aa]]):-atomic([aa,[kk,aa]])

** (1) Retry : p([aa,[kk,aa]],A)
2) p(L_1            ,RL_1):-v(L_1            ,[],RL_1)
*> p([aa,[kk,aa]],A   ):-v([aa,[kk,aa]],[],A   )
_
** (2) Call : v([aa,[kk,aa]],[],A)
2) v([H_1|T_1        ],R_1,L_2):-p(H_1,H1_1),v(T_1        ,[H1_1|R_1],L_2)
*> v([aa |[[kk,aa]]],[] ,A  ):-p(aa ,H1_1),v([[kk,aa]],[H1_1],A  )
_
** (3) Call : p(aa,H1_1)
1) p(X_1,X_1):-atomic(X_1)
*> p(aa ,aa ):-atomic(aa )
** (3) Exit : p(aa,aa)
_
** (4) Call : v([[kk,aa]],[aa],A)
2) v([H_2     |T_2],R_2 ,L_3):-p(H_2     ,H1_2),v(T_2,[H1_2|R_2 ],L_3)
*> v([[kk,aa]|[] ],[aa],A  ):-p([kk,aa],H1_2),v([] ,[H1_2,aa|[] ],A  )
_
** (5) Call : p([kk,aa],H1_2)
1) p(X_2     ,X_2     ):-atomic(X_2     )
*> p([kk,aa],[kk,aa]):-atomic([kk,aa])

** (5) Retry : p([kk,aa],H1_2)
2) p(L_4     ,RL_2):-v(L_4     ,[],RL_2)
*> p([kk,aa],H1_2):-v([kk,aa],[],H1_2)
_
** (6) Call : v([kk,aa],[],H1_2)
2) v([H_3|T_3 ],R_3,L_5 ):-p(H_3,H1_3),v(T_3 ,[H1_3|R_3],L_5 )
*> v([kk |[aa]],[] ,H1_2):-p(kk ,H1_3),v([aa],[H1_3],H1_2)
_
** (7) Call : p(kk,H1_3)
1) p(X_2,X_2):-atomic(X_2)
*> p(kk ,kk ):-atomic(kk )
** (7) Exit : p(kk,kk)
_
** (8) Call : v([aa],[kk],H1_2)
2) v([H_4|T_4],R_4  ,L_6 ):-p(H_4,H1_4),v(T_4,[H1_4|R_4 ],L_6 )
*> v([aa |[] ],[kk],H1_2):-p(aa  ,H1_4),v([] ,[H1_4,kk|[] ],H1_2)
_
** (9) Call : p(aa,H1_4)
1) p(X_3,X_3):-atomic(X_3)
*> p(aa ,aa ):-atomic(aa )
** (9) Exit : p(aa,aa)
_
** (10) Call : v([],[aa,kk],H1_2)
1) v([],R_5      ,R_5      )
*> v([],[aa,kk],[aa,kk])
** (10) Exit : v([],[aa,kk],[aa,kk])
** (8) Exit : v([aa],[kk],[aa,kk])
** (6) Exit : v([kk,aa],[],[aa,kk])
** (5) Exit : p([kk,aa],[aa,kk])
_
** (11) Call : v([],[[aa,kk],aa],A)
1) v([],R_6             ,R_6            )
*> v([],[[aa,kk],aa],[[aa,kk],aa])
** (11) Exit : v([],[[aa,kk],aa],[[aa,kk],aa])
** (4) Exit : v([[kk,aa]],[aa],[[aa,kk],aa])
** (2) Exit : v([aa,[kk,aa]],[],[[aa,kk],aa])
** (1) Exit : p([aa,[kk,aa]],[[aa,kk],aa])

A =  [[aa, kk], aa] ??

yes
```

Key to Spy question 5

| Spy | | EPTB | |
|---|---|---|---|
| j | = | p | |
| p | = | v | |
| 2 | = | aa | |
| 7 | = | kk | |
| X | = | R | } in |
| Y | = | L | } p |
| A | = | H | } |
| B | = | H1 | } |
| X | = | L | } in |
| E | = | X | } j |
| Y | = | RL | } |

FIGURE 11. EPTB Question 5 trace.

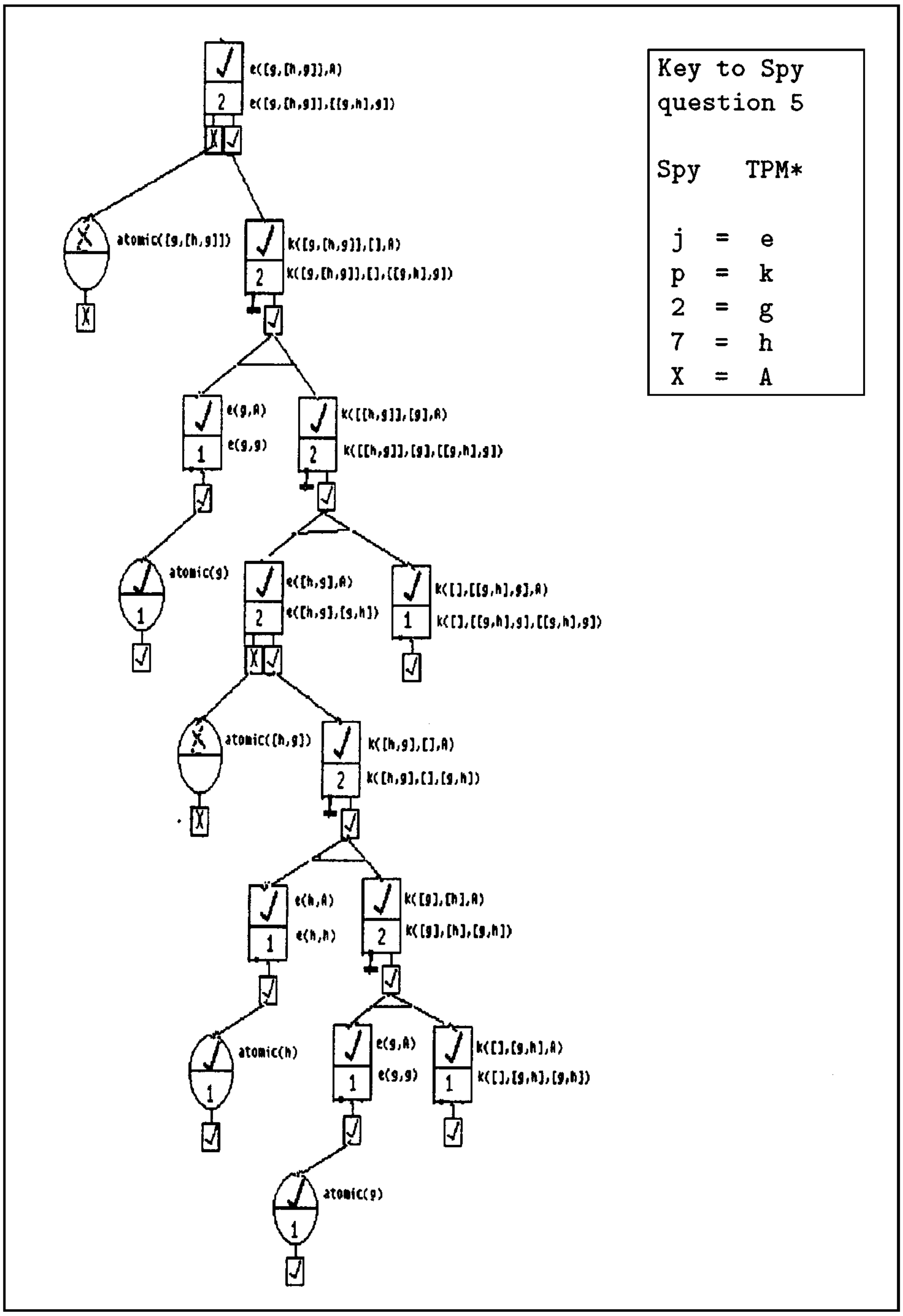


FIGURE 12. TPM* Question 5 trace.

When attempting to solve this problem with the help of the **Spy** trace output subjects have to look through the trace and count up call lines in which either the call term unifies with the fourth clause but with no earlier clauses or the call term was unifiable with both the fourth and fifth clauses, and it can then be deduced that the fourth clause failed for that call.

In the **EPTB** trace, as in question 2 above (which is a less complicated version of this type of problem), subjects can either use the same strategy as for the Spy trace or, more simply, count the lines showing an invocation of the specified clause, indicated in the EPTB format by the display of the clause itself, preceded by an explicit clause number (i.e. `4) mayrphi`...), see Figure 8.

In the TPM* trace version the subjects have to scan clause boxes which explicitly show clause number 4, see Figure 9. They must note that the initial clause box which shows 5 must have matched clause 4 earlier and failed.

Problem 5: This problem is presented with the program code and subjects are asked to pick out a true statement referring to the number of times a particular clause is invoked given a particular query, see Figure 10.

In the **Spy** trace version subjects must systematically scan through the trace identifying how call lines have unified with clauses. The task requires the subject to recognize that certain call lines for "j" refer to more than a single invocation and involve both clauses, see Figure 10.

With the **EPTB** trace it is necessary to find explicit references to the particular clause, recognizing that both a call line and a retry line involve an invocation, see Figure 11.

With the TPM* trace, subjects need to count the number of boxes labelled with that procedure which contain the relevant clause number. They also have to count the number of boxes labelled with the same procedure but with a later clause number with a failed sub-tree corresponding to the relevant clause, see Figure 12.

## 5. Method

### 5.1. DESIGN

The experiment was a within-subjects design: all subjects attempted to solve all the stimuli problems (see Figure 13). Problems were presented in a pseudo-random order; no problem was allowed to be followed immediately by another of the same type but with a different trace. Subjects responded by selecting from multiple choice answers. These choices were randomized for order. The choice of "giving up" with no selection was also

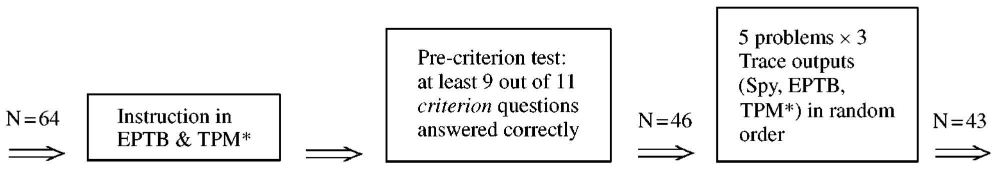

N=64
Instruction in EPTB & TPM*
Pre-criterion test: at least 9 out of 11 *criterion* questions answered correctly
N=46
5 problems × 3 Trace outputs (Spy, EPTB, TPM*) in random order
N=43

FIGURE 13. Experimental design.

provided for each problem. Data on time taken to solve a problem (or give it up) as well as the chosen response were collected.

### 5.2. SUBJECTS

Forty-three undergraduate novice Prolog programmers at Sussex University *fully completed* the problem solving task, and were paid five pounds for taking part in both parts of the experiment. The students were taking a 10 week course in Prolog as part of degrees in computer science or artificial intelligence and had learned at least one other programming language prior to learning Prolog. The students were familiar with the Spy tracer from their Prolog course and learned about EPTB and TPM* as part of the experiment.

### 5.3. PROCEDURE

The instructions, tutorial and problem solving task were presented on Sun workstations in three stages. The entire process was self-administered by the subject, who responded 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 EPTB tracers. Descriptions of various features of both type of trace outputs assumed a basic understanding of Spy traces. Subjects who felt that they had an inadequate understanding of Spy tracers did not participate any further 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 EPTB 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 the main part of the experimental task. Subjects were allowed two attempts to reach this criterion. Following an incorrect response, subjects were given feedback explanations of the correct response. Those who failed to meet the criterion did not take part in the rest of the study. This procedure tried to ensure that only subjects with an adequate understanding of Prolog as well as of TPM* and EPTB were included in the results reported here. Of the original 64 subjects who embarked on the experiment, 18 failed to meet the criterion or withdrew before the next stage.

The third stage was the main problem-solving task. Each problem was presented on one side of the screen as a multiple choice question which the subjects were requested to read through before pressing a key to see the accompanying trace which then appeared on the other side of the screen. This enabled us to collect data on time spent for reading the question separately from time spent in trying to solve the problem with the aid of a trace output. Subjects picked a response (or a "give up" 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. Subjects were asked to complete the task as fast and as accurately as possible. 46 subjects embarked on the test but two failed to finish and one chose the "give up" response to every single question and their data was discarded. This left 43 subjects who completed the problem solving task (though some of these subjects "gave up" on some of the individual problems).

## 6. Results

### 6.1. SOLUTION TIMES

An ANOVA of solution times was carried out with subjects as the random factor and Trace Output (3 levels) and Problem (5 levels) as fixed factors. Solution time was the overall time from first being exposed to the question until the moment that the chosen response was confirmed, see Table 1. There is a significant main effect of trace output, **F**$(2,84) = 11.32$, $p \leq 0.001$, indicating that time required to access relevant information varied between trace outputs. There is a significant main effect of problems, **F**$(4,168) = 42.14$, $p \leq 0.001$. This was expected since problems varied in difficulty.

There is also a significant interaction between trace outputs and problems, **F**$(8,336) = 3.53$, $p \leq 0.01$. Given the interaction, a problem by problem post hoc analysis of simple effects was carried out. Two kinds of comparison were made. (i) A comparison of the mean time of solution for the Spy type trace output compared to the mean of the times for the TPM* and EPTB trace outputs combined. (ii) A comparison of the mean time of solution for the TPM* and EPTB trace outputs.

TABLE 1
*Solution times mean (SD) in seconds of all responses (n = 43)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Trace | 1 | 2 | 3 | 4 | 5 |
| TPM* | 55.2 (23.1) | 87.1 (54.3) | 60.2 (28.9) | 75.3 (54.9) | 131.9 (62.6) |
| Spy | 63.8 (30.2) | 102.7 (58.5) | 77.3 (44.5) | 107.4 (56.3) | 163.9 (100.0) |
| EPTB | 70.4 (34.5) | 75.8 (61.1) | 81.7 (51.9) | 69.7 (36.8) | 112.4 (63.6) |
| Mean | 63.1 (30.1) | 88.6 (58.7) | 73.1 (43.5) | 84.1 (52.4) | 136.1 (79.7) |

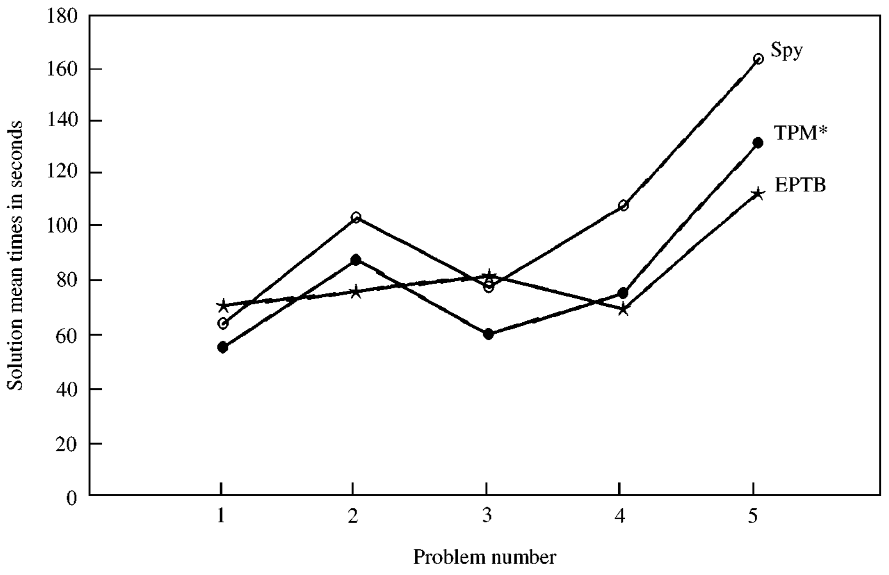


TABLE 2
**F** *values from analysis of simple effects of solution times (n = 43)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Comparison | 1 | 2 | 3 | 4 | 5 |
| Spy vs. (TPM* & EPTB) | 0.04 | 4.82 | 0.86 | 14.93‡ | 8.56† |
| TPM* vs. EPTB | 6.91† | 0.75 | 6.62† | 0.35 | 3.35 |

† $p = 0.05$; ‡$p = 0.01$.

The resultant **F** values are shown in Table 2. In order to contain familywise (FW) error the Scheffé test was used to compute critical values of **F**. The table shows that at $p = 0.05$ Spy type trace outputs produced significantly slower times than combined TPM* and EPTB times for Problems 4 and 5. It also shows that at $p = 0.05$, TPM* trace outputs produced faster solution times than EPTB trace outputs for Problem 1 but slower solution times for Problem 3.

Table 3 illustrates mean solution times of correctly solved problems only. The overall pattern of differences between trace outputs is similar to that in Table 1 which includes all solution times, and on which the solution time ANOVA and analysis of simple effects is based.

### 6.2. RESPONSE ACCURACY

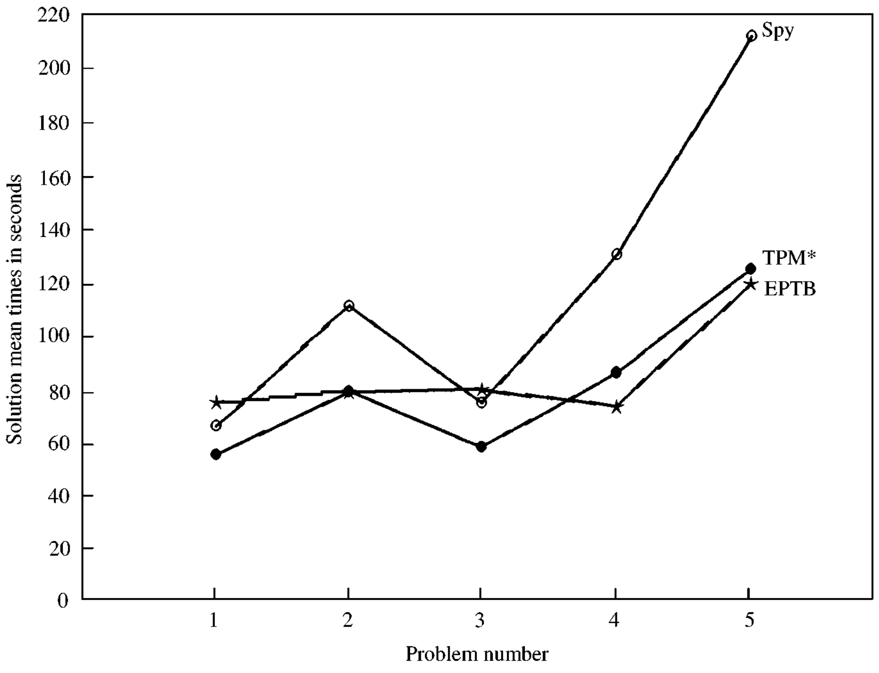
An ANOVA for response accuracy was also carried out. The percentage correct response for each condition is given in Table 4. Given that an individual response was either correct or not, the basic data is essentially binary. Moreover, the overall response accuracy levels in each cell were well over 75% in many cases and under 25% in one case, so the data do not completely conform to the assumptions underlying an analysis of variance. With this proviso in mind, there was a significant main effect of problems, **F**(4,152) = 32.89, $p \leq 0.001$, which reflected the varying level of difficulty of problems. There was a significant main effect of trace outputs, **F**(2,76) = 23.06, $p \leq 0.001$, and there was a significant interaction between trace output and problem, **F**(8,304) = 8.07, $p \leq 0.001$.

Given the interaction, a problem by problem post hoc analysis of simple effects was carried out. Again two kinds of comparisons were made. (i) A comparison of mean response accuracies for the Spy type trace output compared to the mean of the response accuracies for the TPM* and EPTB trace outputs combined. (ii) A comparison of the mean response accuracies for the TPM* and EPTB trace outputs.

The resultant **F** values are shown in Table 5. In order to contain familywise (FW) error the Scheffé test was used to compute critical values of **F**. The table shows that at $p = 0.05$ Spy-type trace outputs produced significantly lower response accuracy than combined TPM* and EPTB response accuracies for Problems 2, 4 and 5. This is a similar result to that for solution times, see Table 2, though the solution time difference for Problem 2 does not reach significance. It also shows that at $p = 0.05$, EPTB trace outputs produced higher response accuracies than TPM* trace outputs for Problems 4 and 5, though this is not mirrored in the response time differences, see Table 2.

TABLE 3
*Solution times mean (SD) in seconds of correct responses only from all subjects*

| | Problem | | | | |
|---|---|---|---|---|---|
| Trace | 1 | 2 | 3 | 4 | 5 |
| TPM* | 56.2 (21.8) | 80.1 (37.2) | 59.3 (21.8) | 87.4 (61.8) | 126.1 (56.6) |
| Spy | 67.0 (28.4) | 112.0 (57.3) | 75.9 (39.0) | 131.4 (93.7) | 212.0 (82.0) |
| EPTB | 76.2 (31.4) | 80.0 (64.8) | 81.1 (46.9) | 74.7 (34.3) | 120.6 (62.0) |
| Mean | 66.3 (28.3) | 88.3 (54.6) | 72.5 (38.5) | 83.4 (51.5) | 140.2 (73.2) |



There is a possibility that problem statements for Problems 4 and 5 were not true isomorphs, see Figure A1 in the appendix. Some problems were couched in terms of "matching" and others in terms of "invocation". In some cases an extra explanatory note was added about the meaning of "invocation".

This extra explanatory sentence appeared in the EPTB version of Problem 4 and in both the Spy and the EPTB version of Problem 5. This may partly account for trend in the data suggesting better performance of Problem 5 compared to Problem 4 in their Spy versions, and for some of the significant disparity between TPM* and EPTB on Problems 4 and 5.

TABLE 4
*Percentage correct response (n = 43)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Trace | 1 | 2 | 3 | 4 | 5 |
| TPM* | 90.7 | 88.4 | 72.1 | 34.9 | 41.9 |
| Spy | 93.0 | 58.1 | 76.8 | 11.6 | 27.9 |
| EPTB | 86.0 | 76.7 | 79.1 | 81.4 | 72.1 |
| Mean | 89.9 | 74.4 | 76.0 | 42.6 | 47.3 |

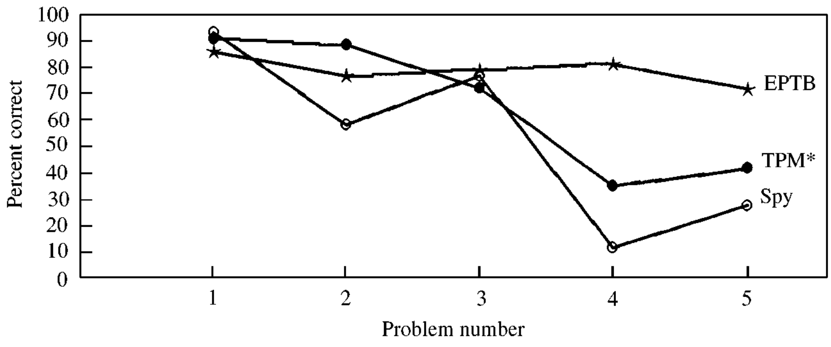


TABLE 5
**F** *values from analysis of simple effects of response accuracy (n = 43)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Comparison | 1 | 2 | 3 | 4 | 5 |
| Spy vs. (TPM* & EPTB) | 0.89 | 9.21† | 0.03 | 34.89‡ | 13.14‡ |
| TPM* vs. EPTB | 0.66 | 2.90 | 0.52 | 30.77‡ | 9.67† |

† $p = 0.05$; ‡$p = 0.01$.

## 6.3. WRONG CHOICES

Problem 1 was answered correctly by most subjects so the following analysis of wrong choices concentrates on Problems 2–5. Table 6 gives the number of times that each of the responses was chosen for each problem and trace. Note that there was a "give up" option in each case and this is shown as option "0" in the table.

Problem 2: In the Spy trace version, eight subjects gave up, six chose option 4 (that "e" is defined by a single recursive clause) and four chose option 2 (that the first clause of "g" has five sub-goals, see Figure 5). None of the Spy subjects chose option 3 (that the second clause of "g" has five sub-goals). In the EPTB version 5 subjects also chose option 4.

TABLE 6
*Choices of each option: ''0'' is ''give up'', starred option is correct*

| | Problem | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | | | | | 3 | | | | | 4 | | | | | 5 | | | | |
| Option | 0 | 1* | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4* | 0 | 1 | 2* | 3 | 4 | 0 | 1* | 2 | 3 | 4 |
| TPM* | 1 | 38 | 1 | 2 | 1 | 3 | 4 | 5 | 0 | 31 | 1 | 20 | 15 | 5 | 2 | 4 | 18 | 4 | 3 | 14 |
| Spy | 8 | 25 | 4 | 0 | 6 | 2 | 5 | 3 | 0 | 33 | 5 | 19 | 5 | 7 | 7 | 4 | 12 | 3 | 7 | 17 |
| EPTB | 2 | 33 | 1 | 2 | 5 | 1 | 1 | 3 | 4 | 34 | 3 | 3 | 35 | 2 | 0 | 5 | 31 | 0 | 5 | 2 |

Problem 3: In both the Spy and the TPM* versions, no subject chose option 3 (that the second clause of "f" is not called), roughly similar numbers divided between options 1 and 2 and giving up. Option 1 states that the first clause of "g" may have more than two sub-goals and option 2 says that the second clause of "g" has exactly three sub-goals. An extra complication in this question was that the subject had to select the option which was false.

Problem 4: To solve this problem, which included the program source code, subjects were asked about the number of times a particular clause is invoked when the program is executed to prove a given query. The correct answer was option 2: Three times, see Figure 7. Table 6 shows that the most frequently chosen incorrect response for both Spy (19) and TPM* (20) was option 1: Once. In Spy the remaining responses were roughly evenly divided amongst the other responses including giving up. In TPM* the next most frequent incorrect choice was option 3: Four times. In EPTB the incorrect choices excluded option 4: Not at all, and divided evenly among the other two and giving up.

Problem 5: This problem was presented with the program code and subjects were asked to pick out a true statement referring to the number of times a particular clause was invoked given a particular query. On average this problem took the longest time to solve with all the trace outputs. More subjects (13) gave up on this problem than on any other.

The most preferred incorrect response for both Spy and TPM* was option 4: "1st clause of j is invoked 3 times". In both cases the other incorrect responses divided fairly evenly among the remaining incorrect options and giving up. In EPTB no subject chose option 2: "1st clause of p is invoked 3 times", and the other incorrect choices were split among the other possibilities.

## 6.4. MODALITY MISMATCHES

In addition to being harder than Problems 1–3, Problems 4 and 5 included the Prolog program as part of the problem statement whereas the simpler problems did not. In order to solve Problems 4 and 5 subjects would have needed to reconcile the program with its trace. In the case of the Spy and EPTB the dominant feature of lines in the trace bear a resemblance to lines in the program. This is rather less than the case with TPM* where the dominant features in the trace are the nodes of the AORTA graph itself, so

TABLE 7
*Percentage correct response for "good" subjects (n = 8)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Trace | 1 | 2 | 3 | 4 | 5 |
| TPM* | 100 | 100 | 75 | 100 | 100 |
| Spy | 87.5 | 62.5 | 100 | 0 | 37.5 |
| EPTB | 87.5 | 100 | 87.5 | 100 | 100 |
| Mean | 91.7 | 87.5 | 87.5 | 66.7 | 79.2 |

there is a modality mismatch between standard Prolog textual programs and their traces. For both Spy and EPTB, search within both the program and the trace is largely up and down the page, whereas in TPM* search needs to follow the links in the AORTA graph. A modality mismatch hypothesis would predict that solution times for Problems 4 and 5 compared to Problems 1–3 would increase to a greater extent for TPM* than for EPTB. It would also predict that response accuracy times for Problems 4 and 5 compared to Problems 1–3 would decrease to a greater extent for TPM* than for EPTB.

The hypothesis is only partially supported by the data. Thus, from Tables 1 and 2 we note that TPM* trace output produces faster solution times than EPTB trace output on Problems 1 and 3 but produces only similar solution times for Problems 4 and 5 (i.e. a relative slowing). From Tables 4 and 5 we note that TPM* trace output produces similar response accuracy to EPTB trace outputs for Problems 1–3 and lower response accuracy for Problems 4 and 5 (i.e. a relative decrease in accuracy). However, comparative solution times results for Problem 2 disrupt this picture.

It could be argued that the subjects were not skilled enough in Prolog and that this combined with the extra difficulty of Problems 4 and 5 to produce poor response accuracy for TPM*. To investigate this issue further, the data on correct solution times was reanalysed. Only data from subjects who successfully solved both Problem 4 and Problem 5 in their TPM* versions were selected. There were eight such "good" subjects and their overall response accuracy is shown in Table 7. These eight subjects all gave the correct answers for Problems 1, 2, 4 and 5 in their TPM* version, for Problems 2, 4 and 5 in their EPTB version and for Problem 3 in the Spy version. None of these eight subjects answered Problem 4 correctly in its Spy version. The correct solution times for these eight subjects are shown in Table 8.

An analysis of simple effects on a problem by problem basis for Table 8 shows that none of the differences in solution times between TPM*-type trace outputs and EPTB-type trace outputs reaches significance under the Scheffé test. The high variance in the data for Problem 2 in its EPTB version is caused by one subject who answered this question very slowly. Excluding this subject from the analysis does not clarify the overall picture. So, overall, the data do not readily support the modality mismatch hypothesis, not least because of the confounding effect of problem difficulty.

TABLE 8
*Solution times mean (SD) in seconds of correct responses only from ''good'' subjects (n = 8)*

| | Problem | | | | |
|---|---|---|---|---|---|
| Trace | 1 | 2 | 3 | 4 | 5 |
| TPM* | 54.1 (17.0) | 71.1 (40.0) | 52.9 (22.6) | 67.4 (28.7) | 142.5 (45.2) |
| Spy | 57.5 (32.9) | 126.0 (51.3) | 64.0 (31.7) | 129.6 (30.0) | 207.2 (71.9) |
| EPTB | 59.5 (32.2) | 101.4 (117.7) | 96.0 (52.8) | 71.0 (48.1) | 104.4 (49.4) |
| Mean | 57.0 (27.2) | 99.5 (77.7) | 71.0 (40.7) | 89.3 (45.6) | 151.4 (69.4) |

## 7. Discussion

Before discussing the likely reasons for the observed results, certain caveats should be restated. The results are based on a non-interactive use of trace outputs for small programs whose traces would fit in a single screen. It is not at all clear how these results could be generalized to much larger programs, to more (natural) interactive debugging, programming and problem solving activities, and therefore no attempt will be made to do so.

To summarize, the results show a major effect of format of trace output on subjects' ability to solve Prolog problems in terms of both the time to solve problems and the accuracy of response.

Overall, as expected, Spy had the most shortcomings. On Problems 4 and 5 it produced longer solution times, and on Problems 2, 4 and 5 it produced lower accuracy than the other two tracers. It did not provide explicit information on system goals, and the simple, non-indented, linear text format is not good at conveying information about backtracking, more complicated retrying of clauses and information about variables involving list manipulation. Much of this implicit information required a large degree of inference on the part of the user. Novice programmers found this very difficult and ended up performing worse than they did with other tracers.

### 7.1. TEXT VERSUS GRAPHICS

Our study illustrates that format and perspective have a significant effect on information access and that the balance of advantage between textual and graphical format is far from clearcut. Graphic TPM* produced faster solution times than textual EPTB on Problems 1 and 3, though response accuracy was similar to EPTB. By contrast, the response accuracy for textual EPTB was better than graphical TPM* for Problems 4 and 5, though the solution times were similar to those for TPM*. On Problem 2 TPM* and EPTB produced similar results.

We have already noted that Problems 4 and 5 were both harder and were presented together with their program code. We investigated the possibility of some modality mismatch effect between lines in the program and "lines" in the trace, but the data were equivocal. So it seems more likely that the solution time and response accuracy disparities are a result of a particular interaction between the problems and the amount of inference required from the trace. These results suggest that hidden state information rather than a modality mismatch was the dominant factor.

### 7.2. HIDDEN STATE INFORMATION

Larkin (1989) draws attention to the problem of hidden state information in a representation. Both Problems 4 and 5 are distinguished from the others in that in each case a single one of the incorrect answer options accounted for a large proportion of the errors in the cases of both Spy and TPM*. It would be necessary to conduct further experiments to establish exactly why subjects chose the options that they did.

A working hypothesis for both Spy and TPM* is that the particular incorrect options were selected because the subjects had to make an inference from the trace notation since the required state information is hidden. The fact that they were novices would have increased the chance of error.† In the case of Spy traces clause matching information is particularly hard to deduce. It is more readily available in TPM* but the subjects still had to deduce that in a clause box showing 5 and an earlier failure, that earlier failure was from a match to clause 4. In both the case of Spy and TPM* that inference is only easily made by understanding the potential unification between the goal in question and the heads of the program clauses.

## 8. Conclusion

This study has shown that representation format has an effect in display-based problem-solving and that the effects are problem-dependent. The study has also shown that the case for graphical representations as compared to textual representation is not absolute but depends, as others have argued, on the nature of the problems-solving being undertaken.

Several lines of development of this work suggest themselves. First, the study should be rerun using a different set of problems. These would be drawn more carefully from programming courses so as to ensure ecological validity and be more carefully disguised to ensure that their statements are isomorphic. The modality mismatch hypothesis needs further work by comparing traces with equivalent hidden state information for problems of equivalent difficulty but offered in different modalities. The analysis of response times, accuracy and particularly errors would benefit from additional protocol "think aloud" data in the manner of Kessler and Anderson (1989).

The study should be extended to the use of tracers as interactive devices and to the use of more realistically sized programs. Some progress has been made on the former, again showing interesting differential effects between tracers and problems (Mulholland, 1994, 1995).

There is scope for further trace notation design. We have made some initial steps in designing and evaluating a Prolog trace notation (TTT) that attempts to marry what we believe are the best features of TPM* and EPTB (Taylor *et al.*, 1991, 1994). The evaluation of TTT with a small number of subjects ($n = 13$) produced broadly similar results to those reported here confirming the relative effectiveness of TPM* compared to Spy (Patel, du Boulay & Taylor, 1994).

† The results also suggest that the initial instruction and the criterion test may not have been sufficiently rigorous.

We thank R. Noble, M. Eisenstadt and the referees for most helpful comments. 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.

## References

BRNA, P., BRAYSHAW, M., BUNDY, A., DODD, T., ELSOM-COOK, M. & FUNG, P. (1991). An overview of Prolog debugging tools. *Instructional Science*, **2/3,** 193–214.

BYRD, L. (1980). Understanding the control flow of Prolog programs. In S. TARNLUND, Ed. *Proceedings of the Logic Programming Workshop*, pp. 127–138.

CLOCKSIN, W. F. & MELLISH, C. (1981). *Programming in Prolog*. Berlin: Springer.

CUNNIFF, N. & TAYLOR, R. (1987). Graphical versus textual representation: an empirical study of novice's program comprehension. In G. OLSON, S. SHEPHERD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corporation.

DAVIES, S. P. (1993*a*). Expertise and display-based strategies in computer programming. In J. ALTY, D. DIAPER & S. GUEST, Eds. *Proceedings of HCI'93*, pp. 411–423. Cambridge: Cambridge University Press.

DAVIES, S. P. (1993*b*). Models and theoreies of programming strategy. *International Journal of Man–Machine Studies*, **39,** 237–267.

DICHEV, C. & DU BOULAY, B. (1989). An enhanced trace tool for Prolog. In *Proceedings of the 3rd International Conference, Childen in the Information Age*, pp. 149–163, Sofia, Bulgaria.

DU BOULAY, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, **2,** 57–73.

EISENSTADT, M. (1984). A powerful Prolog trace package. In *Proceedings of the 6th European Conference on Artificial Intelligence (ECAI-84)*. Amsterdam: North-Holland.

EISENSTADT, M. & BRAYSHAW, M. (1988). The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, **5,** 277–342.

EISENSTADT, M., BRAYSHAW, M. & PAINE, J. (1991). *The Transparent Prolog Machine*. Oxford: Intellect.

EISENSTADT, M. & DIXON, M. (1988). *Intensive Prolog Workbook*. Milton Keynes: Open University Press.

GEGG-HARRISON, T. (1991). Learning Prolog in a schema-based environment. *Instructional Science*, **20,** 173–192.

GILMORE, D. J. & GREEN, T. R. (1988). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology*, **40A,** 423–442.

GREEN, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, **50,** 93–109.

GREEN, T. R. G. (1989). Cognitive dimensions of notations. In A. SUTCLIFFE & L. MACAULAY, Eds. *People and Computers V Proceedings of HCI'89*, pp. 443–60. Cambridge: Cambridge University Press.

GREEN, T. R. G. (1991). Describing information artifacts with cognitive dimensions and structure maps. In D. DIAPER & N. V. HAMMOND, Eds. *People and Computers VI: Usability Now!, Proceedings of HCI'91*. Cambridge: Cambridge University Press.

GREEN, T. R. G., BELLAMY, R. K. & PARKER, J. (1987). Parsing and gnisrap: a model of device use. In G. OLSON, S. SHEPHERD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop*, pp. 132–146. Norwood, NJ: Ablex Publishing Corporation.

GREEN, T. R. G., PETRE, M. & BELLAMY, R. K. (1991). Comprehensibility of visual and textual programs: a test of superlativism against the "match-mismatch" conjecture. In J. KOENEMANN-BELLIVEAU, T. MOHER & S. ROBERTSON, Eds. *Empirical Studies of Programmers: Fourth Workshop*, pp. 121–146. Norwood, NJ: Ablex Publishing Corporation.

HOOK, K., TAYLOR, J. & DU BOULAY, B. (1990). Redo "try once and pass": the influence of complexity and graphical notation on novices' understanding of Prolog. *Instructional Science*, **19,** 337–360.

KESSLER, C. & ANDERSON, J. R. (1989). Learning flow of control: Recursive and iterative procedures. In E. SOLOWAY & J. SPOHRER, Eds. *Studying the Novice Programmer*. Hillsdale, NJ: Erlbaum.

LARKIN, J. H. (1989). Display-based problem solving. In D. KLAHR & K. KOVOSTSKY, Eds. *Complex Information Processing: The Impact of Herbert A. Simon*, pp. 319–341. London: Erlbaum.

LARKIN, J. H. & SIMON, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, **11,** 65–99.

LOOI, C. (1991). Automatic debugging of Prolog programs in a Prolog intelligent tutoring system. *Instructional Science*, **20,** 215–263.

MANN, L. M., LINN, M. C. & CLANCY, M. (1994). Can tracing tools contribute to programming efficiency? the LISP evaluation modeller. *Interactive Learning Environments*, **4,** 96–109.

MELLISH, C. (1984). Teach * tracer, part of University of Sussex Poplog Programming environment.

MULHOLLAND, P. (1994). The effect of graphical and textual visualisation on the comprehension of Prolog execution by novices: an empirical analysis. In *6th Workshop of the Psychology of Programming Interest Group*, pp. 18–26. Open University.

MULHOLLAND, P. (1995). *A framework for describing and evaluating software visualisation systems: a case-study in Prolog*. Ph.D. Thesis, Knowledge Media Institute, Open University.

PAIN, H. & BUNDY, A. (1987). What stories should we tell novice Prolog programmers?. In R. HAWLEY, Ed. *Artificial Intelligence Programming Environments*. Chichester: Ellis Horwood.

PATEL, M. J., DU BOULAY, B. & TAYLOR, C. (1991*a*). Effect of format on information and problem solving. In *Proceedings of 13th Annual Conference of the Cognitive Science Society*, pp. 852–856, Chicago.

PATEL, M. J., DU BOULAY, B. & TAYLOR, C. (1991*b*). Prolog tracers and information access. In L. PRESS & J. GORNOSTAEV, Eds. *Proceedings of the First Moscow HCI'91 Workshop*, pp. 140–145. Moscow: International Centre for Scientific and Technical Information.

PATEL, M. J., DU BOULAY, B. & TAYLOR, C. (1994). Textual tree (Prolog) tracer: an experimental evaluation. In D. GILMORE, R. WINDER & F. DETIENNE, Eds. *User-Centred Requirements for Software Engineering Environments*, pp. 127–141. Berlin: Springer.

PETRE, M. & GREEN, T. R. G. (1990). Where to draw the line with text: some claims by logic designers about graphics in notation. In D. DIAPER, Ed. *Proceedings of Human–Computer Interaction INTERACT'90*, pp. 463–468. Amsterdam: North-Holland.

PRICE, B. A., BAECKER, R. M. & SMALL, I. S. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, **4,** 211–266.

RAJAN, T. (1991). An evaluation of APT: an animated program tracer for novice programmers. *Instructional Science*, **20,** 89–110.

SCAIFE, M. & ROGERS, Y. (1996). External cognition: How do graphical representations work?. *International Journal of Human-Computer Studies*, **45,** 185–213.

SHU, N. (1988). *Visual Programming*. New York: Van Nostrand Reinhold.

TAYLOR, C., DU BOULAY, B. & PATEL, M. J. (1991). Outline proposal for a Prolog 'Textual Tree Tracer' (TTT). *Cognitive Sciences Research Paper 177,* University of Sussex, Brighton, UK.

TAYLOR, C., DU BOULAY, B. & PATEL, M. J. (1994). Textual tree trace notation for Prolog: an overview. In R. M. BOTTINO, P. FORCHERI & M. T. MOLFINO, Eds. *International Conference on Logic Programming ICLP'94: Post-Conference Workshop on Logic Programming and Education*, Santa Margherita Ligure.

Paper accepted for publication by Associate Editor, Professor E. A. Edmonds.

## Appendix

This section includes TPM* and EPTB questions (Figure A1) for Question 4 to illustrate the degree of disguise between question isomorphs. The correct answer choice is starred.

```
Given the program below, and the trace output shown, how
many times is the 4th clause invoked (whether successfully
or unsuccessfully) during the execution of the following
goal (N.B. A clause is "invoked" if its head matches a call -
it will then succeed if all its subgoals succeed, and fail
otherwise.)

?- mayrphi([9,19], [1,9,11,30], L).

mayrphi([], _, []).
mayrphi(_, [], []).
mayrphi([H1|T1], [H1|T2], [H1|Others]):-
    mayrphi(T1, T2, Others).
mayrphi([H1|T1], [H2|T2], Others):-
    H1 < H2, mayrphi(T1, [H2|T2], Others).
mayrphi([H1|T1], [H2|T2], Others):-
    H1 > H2, mayrphi([H1|T1], T2, Others).

1. Once
2. Three times*
3. Four times
4. Not at all
```

```
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).

1. One
2. Three*
3. Four
4. None
```

FIGURE A1. Question 4 EPTB (top), TPM* (below).