Incorporating Software Visualization into Prolog teaching: a challenge, a restriction, and an opportunity

Paul Mulholland

Knowledge Media Institute, The Open University

Abstract: The difficulties students have in learning and using Prolog are well documented (e.g. Taylor, 1988). Many of these difficulties are due to the complexity of the execution model (Fung et al, 1990). Ever since the Byrd Box model (Byrd, 1980), the challenge has been to present the execution model in the most effective way. The term Software Visualization has been coined to describe the process of using textual and/or graphical formalisms to describe the execution of computer programs. Software Visualizations for Prolog have been numerous, though empirical evaluation has been sparse. Recent empirical work using a new empirical framework (Mulholland, 1995) has found that Prolog Software Visualizations differ markedly in how students can use them, and that even minor changes to the way the execution is presented can have dramatic effects on how well it is understood by students.

Despite the large amount of systems that have been developed to clearly visualize the execution model, their educational effectiveness without teacher supervision appears to be seriously restricted. Students were often found to reinterpret the display to fit their own misunderstandings, rather than learning from what the visualization was showing them. Software Visualizations do little to lessen the role of the human teacher.

The development of the World Wide Web, though provides a new opportunity for Software Visualization within Prolog tuition: as a rich communication medium in which to describe difficult concepts. We are currently exploring this idea with the Internet Software Visualization Laboratory (Domingue and Mulholland, 1997), which allows visualizations of programming concepts to be communicated between students and teacher, using either electronic movies or synchronous tutorials.

1. Introduction

Software Visualization (SV) is the process of using techniques such as typography, graphic design, animation and cinematography to provide representations of a program and its execution. As Prolog has a particularly complex execution model, which students have difficulty understanding, a number a SVs (or tracers) for Prolog have been developed. Each of these SVs has associated claims put forward by the designers, explaining how their system can help Prolog students. Though due to a lack of empirical evaluation it is often difficult to determine what progress is being made in clearly presenting the Prolog execution model.

This paper presents the ISM (Information, Strategy Misunderstanding) Framework which provides a principled approach to the challenge of how an SV can best present Prolog to a student. The next section describes the ISM framework. The following sections describe an empirical study using the ISM framework to find out how students are able to use four existing Prolog SVs. This is followed by the design of a new Prolog SV, based upon the lessons learnt from the empirical study. This is crucially important as it illustrates how this framework can be used to successfully underpin the challenge to build more and more effective SV tools for Prolog students.

Following on from this, some observations are made from the empirical work as to what extent SV can really deliver as an educational tool. It has been widely thought that SV could be used by students for learning by free exploration. There are serious restrictions as to how much students can learn this way due to their tendency to reinterpret what the SV is showing them to fit their own misunderstandings. The final section though takes up the challenge of how SV can be effectively used within an educational setting in a way which makes use of its strengths, though is still integrated with the expertise of the teacher.

2. The ISM (Information, Strategy, Misunderstanding) Framework

An SV for Prolog, or any other language, is used to help someone to comprehend how some particular program or programming construct works. The effectiveness of an SV is the extent to which it supports the learner in this comprehension process. A review of the psychology of programming literature (Mulholland, 1995) identified three kinds of "cognitive evidence" which should strongly relate to how well an SV helps someone to understand a program: (i) the extent to which useful kinds of information can be accessed from the SV, (ii) to extent to which comprehension strategies can be affective applied to what the SV is presenting, and (iii) the extent to which the way the SV presents the execution can be misunderstood by the student.

2.1 Information

Some studies have already considered the kinds of information that are accessed during program comprehension phases. For example, Bergantz and Hassell (1991) used protocol analysis to measure information access during Prolog comprehension using four main information types (which they termed relations): control flow, data flow, program structure and program function. Clearly these information types will have to be extended and modified to take into account the incorporation of the SV, though these outline some basic information types that are central to program comprehension and would therefore be expected during SV use.

2.2 Strategies

Some work has also looked at the use of programming strategies during program comprehension and debugging. For example, Green (1977) considered the ease with which forward and backward reasoning strategies can be utilised given different control constructs within the program. It is therefore possible that forward and backward reasoning strategies could be employed when using an SV during program comprehension. More specific strategies relating to SVs may also be observed.

2.3 Misunderstandings

A great deal of work has been carried out on the kinds of misunderstandings novices tend to have of the Prolog execution model (Fung et al, 1990). The nature of the SV may affect the number and kind of misunderstandings the student has of the execution. It would be expected that using a SV would reduce many of the well documented misunderstandings though the extra demands of dealing with the SV could create new misunderstandings which have not previously been investigated.

3. Challenge Part I: Evaluating existing Prolog SVs

This section will briefly present the four SVs to be evaluated. The nature of the study will then be described. The results are then presented, encompassing: overall performance on the task, information access, strategy utilisation, and the prevalence of misunderstandings.

3.1 The SVs

Four SVs were used in the study, which are described below. These were implemented within the same environment, the Prolog Program Visualization Laboratory (PPVL) (Mulholland, 1995). This provided an experimental laboratory allowing a number of fully implemented SVs to be compared within the same environment. PPVL provides similar interface and navigation for each SV and records all user activity at the terminal.

Spy (Byrd, 1980) (figure 1, top left) is a stepwise, linear, textual SV system which adopts the Byrd Box model of Prolog execution. Spy gives a basic procedural account of the execution. The head of a clause can be thought of as a procedure and the tail treated as one or more sub procedures. Byrd's aim in the development of Spy was to provide a basic but complete account of Prolog underpinned by a consistent execution model.

PTP (Prolog Trace Package) (figure 1, top right) was developed by Eisenstadt (1984) to provide a more readable and detailed account of Prolog execution than is found in Spy. PTP aimed to make the account of execution as explicit as possible, thereby reducing the amount of interpretation required by the user. Particular areas where PTP aimed to improve on Spy was in the presentation of more specific status information and a more explicit presentation of how the program relates to the execution.

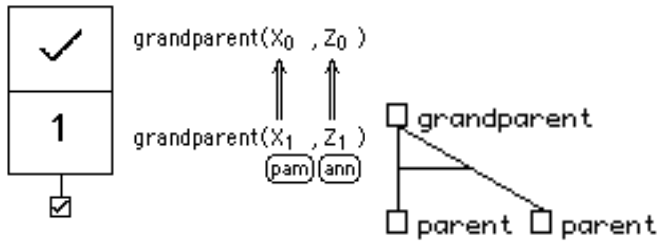
Spy

```
 call  grandparent(_1, _2)
 UNIFY 1   []
   call  parent(_1, _3)
   UNIFY 1   [_1 = tom _3 = liz]
   exit  parent(tom, liz)
   call  parent(liz, _2)
   fail  parent(liz, _2)
   redo  parent(tom, liz)
   UNIFY 2   [_1 = pam _3 = bob]
   exit  parent(pam, bob)
   call  parent(bob, _2)
   UNIFY 3   [_2 = ann]
   exit  parent(bob, ann)
 exit  grandparent(pam, ann)
```

PTP

```
  1: ? grandparent(_1, _2)
  2: > grandparent(_1, _2) [1]
  3:   ? parent(_1, _3)
  4:   +*parent(tom, liz) [1]
  5:   ? parent(liz, _2)
  6:   -~parent(liz, _2)
  7:   ^ parent(tom, liz)
  8:   < parent(tom, liz) [1]
  9:   +*parent(pam, bob) [2]
 10:   ? parent(bob, _2)
 11:   +*parent(bob, ann) [3]
 12: + grandparent(pam, ann) [1]
```

TPM



TTT

```
>>>1: grandparent(X, Z)   1S
|1    X = pam
|     Z = ann
 ***2: parent(X, Y_1)   1SF/2S
 |1    X ≠ tom
 |     Y_1 ≠ liz
 |2    X = pam
 |     Y_1 = bob
 ***3: parent(liz, Z)   Fm
 ***4: parent(bob, Z)   3S
 |3    Z = ann
```

Figure 1. Software visualization displays of a Prolog grandparent program, which infers grandparent relations from a given set of parent relations. The SVs shown are (top left) the linear textual Spy, (top right) the readable textual PTP, (bottom left) the graphical TPM showing the AND/OR tree and the fine-grained details of the grandparent node, and (bottom right) the non-linear textual TTT.

TPM (Transparent Prolog Machine) (Brayshaw and Eisenstadt, 1991; Eisenstadt and Brayshaw, 1988) (figure 1, bottom left) aimed to provide the very detailed account provided by PTP in a much more accessible form. TPM uses an AND/OR tree model of Prolog execution. Execution is shown as a depth first search of the execution tree. Unlike the other SVs, TPM incorporates two levels of granularity. An AND/OR tree model is used to provide an overview of the execution. Fine grained views giving details relating to a particular goal within the program execution are obtained by selecting the node in question.

TTT (Textual Tree Tracer) (Taylor, du Boulay and Patel, 1991) (figure 1, bottom right) has an underlying model similar to TPM but uses a sideways textual tree notation to provide a single view of execution which more closely resembles the source code. Unlike linear textual SVs such as Spy and PTP, current information relating to a previously encountered goal is displayed with or over the previous information. This keeps all information relating to a particular goal in the same location. The aim behind TTT was to provide the richness of information found in the TPM SV in a form more closely resembling the underlying source code. This approach necessitates an important trade-off in the design of the SV notation. TPM aims to show the structure and nature of the execution by using a graphical formalism to show an overall picture. For TTT, constructing a SV notation with a close affinity to the underlying source code was a primary aim.

3.2. The experiment

A four way between subjects design was used with 16 novice Prolog programmers per cell working in pairs. Each pair of subjects were given five minutes to familiarise themselves with a program presented on a printed sheet. They each retained a copy of this program throughout the experiment. The program was an isomorphic variant of the one used by Coombs and Stell (1985) to investigate backtracking misconceptions. They were then asked to work through the traces of four versions of the program which had been modified in some way. Their task was to identify the difference between the program on the sheet and the one they were tracing. They had no access to the source code of the modified versions. After five minutes the subjects were given the option to move onto the next problem. This was used as an upper bound for timing data. Verbal protocols were taken throughout.

Program modifications were selected which required the novice to focus on different types of information in order to correctly identify the change. The four problems given were a change in a relation name, a changed atom name, a data flow change and a control flow change. The data flow change was either passing the wrong variable from a rule or changing a variable within a rule to an atom. The control flow change was either a swap in the subgoal order of a rule or the fact order within the database.

3.3 Overall performance

The mean number of problems solved by each subject pair in total are shown in figure 2. PTP performed the best overall. The lowest success rate was found with the graphical TPM. There was a significant main effect for SV, $F(3, 28) = 3.260$, $p < 0.05$.

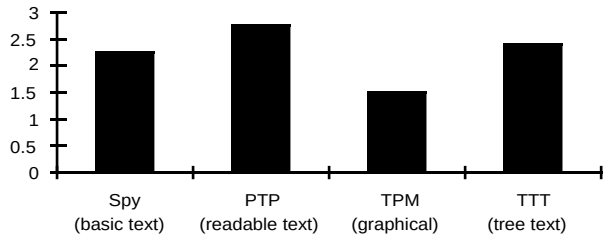


Figure 2. Mean number of problems completed within five minutes.

3.4. Information

A preliminary analysis of the protocols identified the kinds of information referred to by subjects. Three information types identified important differences between the SVs. These were control flow information (CFI), data flow information (DFI) and SV related information (SVI). Control flow statements may refer to the order in which events happen within the execution or refer to the status of goals within the program or the clause number within the program with which a goal has unified. Data flow statements either referred to bindings occurring within a goal or to the sharing of values between variables contained in different goals. SVI utterances related to the navigation or notation of the SV, usually relating to which button has to pressed or referring to some symbol within the notation.

The means for CFI, DFI and SVI information by SV are shown in figure 3. CFI and DFI are successful attempts to access information. SVI represents attempts to access information about the SV rather than the program itself.

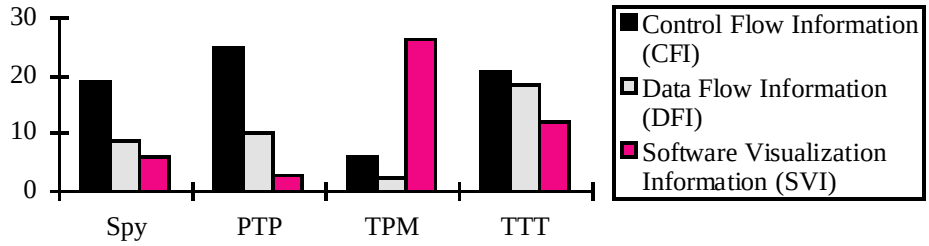


Figure 3. Mean number of CFI, DFI and SVI utterances.

As control flow and data flow are the more central information types to program comprehension, a two way mixed ANOVA was performed on this data. This revealed significant main effects for SV, $F(3, 26) = 5.155$, $p < 0.01$ and information type, $F(1, 26) = 15.262$, $p < 0.01$. A one way analysis of variance revealed a significant main effect for SV related utterances (SVI), $F(3, 26) = 5.536$, $p < 0.01$.

3.5. Strategies

A preliminary analysis of the protocols identified the kinds of comprehension strategies used by subjects. Five comprehension strategies showed interesting differences between the SVs. These were review control flow (REVIEW CF), review data flow (REVIEW DF), test control flow (TEST CF), test data flow (TEST DF), and mapping between the SV and the code (MAPPING).

The strategy REVIEW CF denotes reviewing some or all of the previous steps in the control flow in order to better understand the current state of the execution. A similar strategy was concerned with reviewing previous changes in data flow in order to understand or explain all or any of the current variable bindings (REVIEW DF). The mean occurrence of review strategies is shown in figure 4.

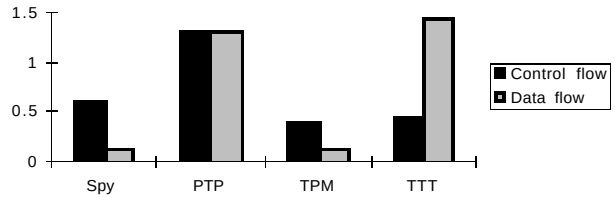


Figure 4. Mean number of control flow and data flow review strategies (REVIEW CF and REVIEW DF).

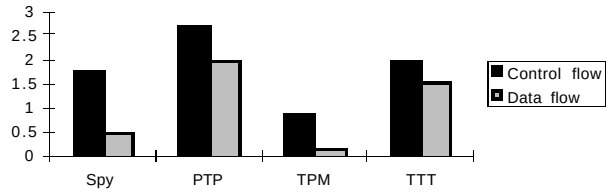


Figure 5. Mean number of control flow and data flow test strategies (TEST CF and TEST DF).

A two factor ANOVA comparing the four SVs across the two review strategies revealed a main effect for SV, $F(3, 26) = 3.495$, $p < 0.05$ and a significant interaction between SV and strategy, $F(3, 26) = 4.304$, $p < 0.05$.

Test strategies related to the prediction and testing of future control flow (TEST CF) or data flow (TEST DF) states. When using this strategy the subjects would make some prediction as to the a future control flow or data flow state and then step forward to test their prediction. The distribution of test strategies is shown in figure 5. A two factor ANOVA comparing the four SVs across the two test strategies revealed a main effect for SV, $F(3, 26) = 3.253$,$p < 0.0378$.

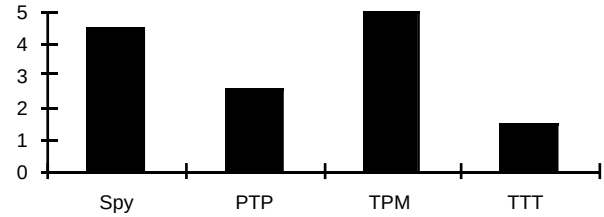


Figure 6. Mean number of MAPPING strategies.

Another strategy identified the mapping made between the SV and the program code (MAPPING). Often it was not clear what level of understanding was occurring during the use of this strategy. In its simplest form, the subjects could search for the physical resemblance between some text in the SV and some text appearing in the source code. Subjects using this strategy sometimes were able to identify a similarity between the code and SV but were still unclear as to what was occurring. The mean number of MAPPING strategies for each SV are shown in figure 6. A one way analysis of variance of the distribution of the MAPPING strategy revealed a main effect for SV, $F(3, 26) = 5.656$, $p < 0.01$.

3.6. Misunderstandings

A preliminary analysis of the protocols revealed four main misunderstandings of the SV. These were: making an inappropriate mapping between the visualization and the program (MAPM), deriving an incorrect model of either control flow (CFM) or data flow (DFM), or failing to appreciate the stage within the execution being presented, referred to as Timing Misunderstandings (TM). The mean number of misunderstandings per subject pair are shown in figure 7.

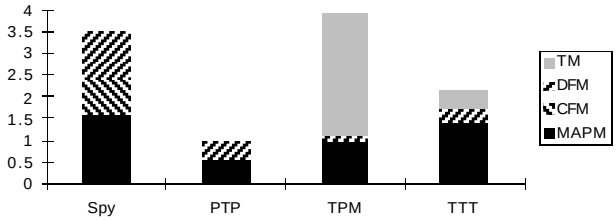


Figure 7. Mean number of each misunderstanding per subject pair.

MAPM (Mapping Misunderstanding) refers to when part of the program is compared in an inappropriate way to a part of the SV. Essentially, this misunderstanding is an inappropriate application of the MAPPING strategy described earlier. The MAPM misunderstanding occurred, when either some part of the display and code were incorrectly compared because of some surface similarity, such as the presence of a particular word, or conversely, when some part of the code and display were thought to be unrelated because of some surface dissimilarity such as the use of different variable names in the display.

CFMs (Control Flow Misunderstandings) resulted from either the subjects adopting an ad hoc false interpretation of control flow which the SV failed to counteract, or an incorrect interpretation of the SV which lead subjects to reject their correct model of control flow. Similarly, DFMs (Data Flow Misunderstandings) tended to result from a disparity between the subjects' assumption regarding what should happen at some point in terms of data flow, and their interpretation of what was really happening, derived from the display. Once again, the origin of the misunderstanding seemed to be either the subjects adopting an incorrect model of data flow on an ad hoc basis which the SV failed to counteract, or a false interpretation of the display leading them to reject their correct assumptions.

Time misunderstandings (TM) were those resulting from a failure to appreciate the position in the execution being shown at some particular point. For example, some subjects thought the SV was faulty because a particular variable was unbound though the stage of the execution at which it becomes bound had not yet occurred.

The mean number of misunderstandings per subject pair are shown in figure 7. A one way ANOVA of the total number of misunderstandings for each subject pair revealed a main effect for SV, $F(3, 26) = 3.669$, $p < 0.05$.

4. Challenge Part 2: Designing a new Prolog SV

Further SVs were designed using the results of the empirical study described above, among them Plater. The design of Plater was underpinned by four design decisions derived from an interpretation of the empirical data. An account of precisely how the design decisions were derived can be found elsewhere (Mulholland, 1995; to appear). The interpretation focused on how useful kinds of information access and strategy utilisation could be encouraged and how misunderstandings could be reduced. The design decisions were: (i) supporting an effective mapping between the SV and the source code, (ii) presenting data flow in a way which remains faithful to the source code, (iii) providing a clear perspective on the execution history in a way which allows students to understand what has already happened, what is currently happening, and what is about to happen, and (iv) clearly showing how the execution resumes when a goal fails. These four issues and how each are tackled in the Plater design will be considered in turn.

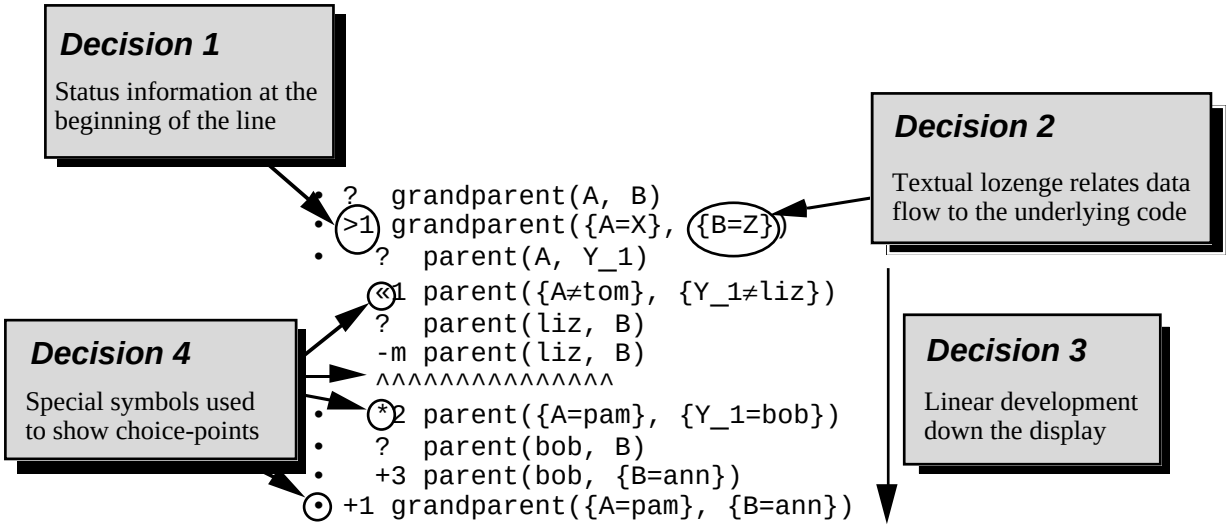


Figure 8. Plater visualization of the grandparent program with the design decisions identified.

First, the issue of helping the mapping in terms of control flow was tackled by moving all information relating to the status of the goal and how that relates to the code to the beginning of each line to encourage students to take note of it before moving onto analysing the goal itself. The intended outcome of this change is a reduction in the number of mapping misunderstandings (MAPM). These misunderstandings tended to arise when the subject lost the program context for the goals shown in the SV and made an inappropriate mapping between the two. This change would in turn increase access to control flow information and its utilisation in comprehension strategies. In figure 8, “Decision 1” shows how in line two of the visualization, the information that the first clause has been entered is shown near the beginning. This is true of all lines in the visualization which relate to a goal.

Second, it is important that the SV presents data flow information in way that does not prevent the user from mapping to the variables as shown in the program. This is dealt with in Plater by the use of textual lozenges whereby the left variable in each lozenge reflects data flow and the right variable is faithful to the underlying source code. In figure 8, “Decision 2” shows how the top level variable “B” is permeating through the execution, and is related to variable “Z” within the source code. Presenting both the data flow and code faithful variables in context is intended to reduce the mapping and data flow misunderstandings and replace these with effective mapping and data flow strategies.

Third, a clear indication of how far through the execution the SV has progressed is illustrated by the length of the display produced by the SV. Each event with the execution causes the SV to extend by one line. This should ensure that the students are not prone to time misunderstandings, and can appropriately access and utilise control flow information. Students should also be more able to apply test strategies. Clearly, the application of test strategies will be inhibited if students are unclear as to what is the current state of the execution.

Fourth, an attempt to tackle the problem of encouraging a correct understanding of control flow during backtracking was done by adopting a choice-point model of execution (explained in Byrd, 1980) which shows explicitly to where the execution will backtrack, should the current execution path fail. A choice-point execution model can be thought of as finding a route through a maze. When alternative paths are available, this is marked as a choice-point. Should the current path fail, the search continues by returning to the choice-point and taking another available path. Within Plater, four notational conventions are required to show the choice-point model. In figure 8, “Decision 4” identifies these. The important concepts required for a choice-point model are a knowledge of available choice-points, choice-points that have been “used-up”, points of failure and steps of the current successful path. These are shown in Plater as *, «, ^^^^^^^^^^^^^^^ and • respectively. If this design decision is successful, the occurrence of control flow and mapping misunderstandings during complex control flow sequences should reduce and be replaced with effective control flow and data flow strategies. In a further empirical study, Plater was found to be a more effective tool for Prolog novices than the four SVs previously studied (see Mulholland, 1995, to appear) for details, illustrating how the ISM framework can be used to feed the results of an evaluation into a new design.

5. Restriction: The Problem of Reinterpretation

Two main approaches have been taken to the issue of how software can be used to support computer programming education: Intelligent Tutoring Systems (ITSs) and SV. ITS has lost favour in recent years due to its practical difficulties (Eisenstadt, Price, and Domingue, 1992). The range of conceptual bugs found even in relatively small student programs is vast. Cataloguing all these bug variations is an immense task. Secondly, even if the bug can be spotted by the ITS it is not clear what kind of response should be provided. The student may not fully appreciate the description given by the ITS and fail to overcome the problem.

This kind of issue lead to a shift in focus toward SV as a more promising solution. SV aims to provide a clear story of the language which the student can explore and learn from. There is therefore no need to develop bug catalogues or consider what kind of response should be given to the student when a bug is located, the central issue for SV being the clarity of the external representation used to present the language and its execution.

SV though has difficulties of its own (Mulholland and Eisenstadt, in press). One serious problem is the students' tendency to reinterpret the display to fit their own expectations. For example, often when students had a misunderstanding of the program execution, the display would be interpreted in such a way as to be consistent with their false expectations, rather than the display leading the students to challenge and reformulate their understanding. It was hoped that SV would succeed in providing an environment whereby students could question and reformulate their own understanding. Though this may occur, it is certainly not always the case. This suggests that SV cannot be relied upon as a stand-alone educational tool and must be carefully incorporated into, and supported by other aspects of the educational environment.

Even though SV can help students in their understanding of computer programs, it does not lessen the vital role of human teacher. An important area in which the SV could be used to support the work of the teacher as a communication aid. We are currently exploring this idea with the Internet Software Visualization Laboratory, described in the next section.

6. Opportunity: SV for Communication

The Internet Software Visualization Laboratory (ISVL) (Domingue and Mulholland, 1997; to appear), supports individual exploration, and both synchronous and asynchronous communication. As a single user, students are able to explore a program on their own, though ISVL can also be used as a synchronous communication medium whereby the tutor can provide an annotated demonstration of a program and its execution. Finally, ISVL can be used to support asynchronous communication by allowing the tutor to prepare short educational movies for students to view when convenient (figure 9). The ISVL environment runs on a conventional web browser and is therefore platform independent, has modest hardware and bandwidth requirements, and is easy to distribute and maintain.

ISVL is currently being used on a distance education Prolog course, though could also play a extremely valuable role within a conventional university setting. Within a conventional university, ISVL will allow tutors build an archive of useful educational resources, and permit students to develop an interactive portfolio of their programming achievements. An interesting possibility would be that prospective employers may be able to view a students' portfolio of ISVL movies, as a measure of their ability to communicate their work, in the same way that an artist may take a portfolio of work to an interview.

ISVL is of even greater importance within a distance education setting, as it allows students for the first time to work collaboratively on programming projects. The ability to communicate and work in teams is becoming considered an ever more important aspect of computer science courses. Team work and the clear presentation of ideas more accurately reflects the current tasks of the industrial programmer than sitting alone carrying out a programming assignment (Dawson, Newsham and Kerridge, 1992). ISVL will allow investigation of how distance education courses can provide more scope for collaborative and group activities within the curriculum. We also hope that the ISVL framework will in the future be adopted within an industrial context, whose teams of programmers and interested parties often need the types of communication offered by ISVL, and a simple way of recording successive developments of the software.

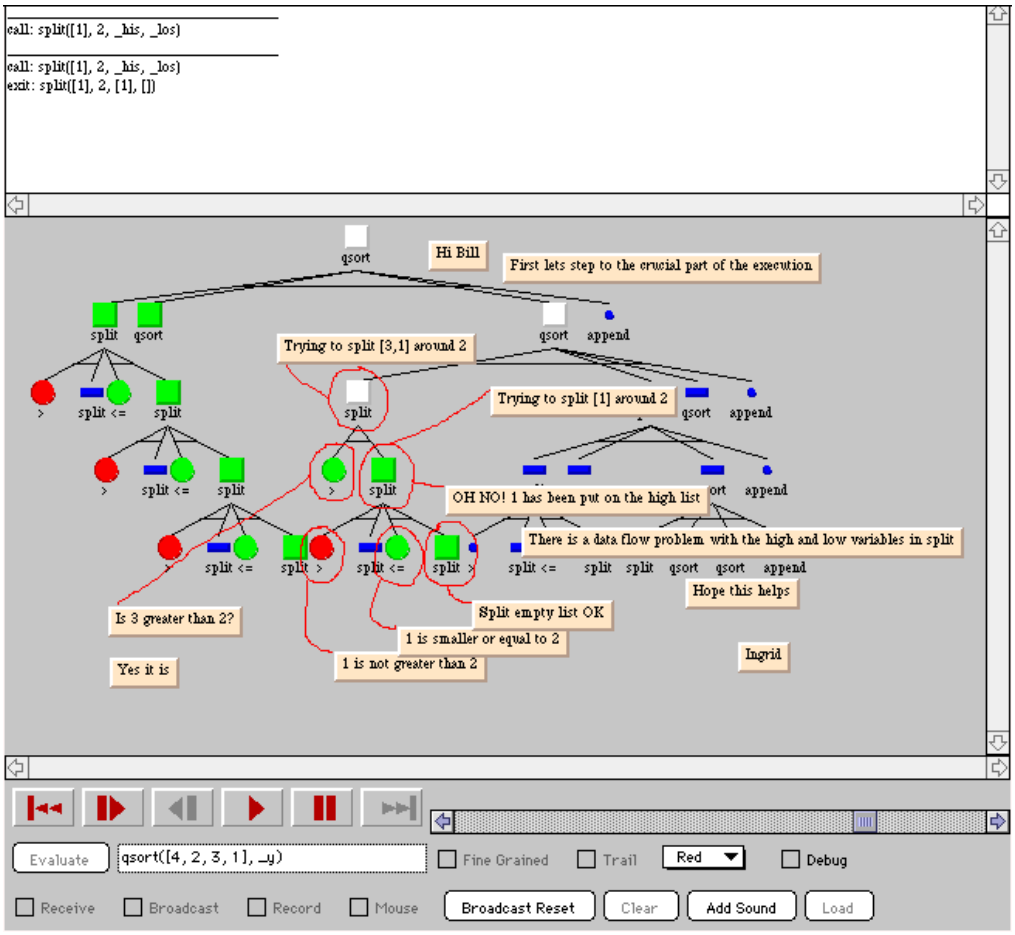


Figure 9. An ISVL movie prepared by a tutor, showing a student how to go about fixing their buggy program

7. Summary

This paper described how the challenge of effectively presenting the Prolog execution model to students can be underpinned using the ISM Framework, by evaluating existing SVs in way which can motivate future designs. It was noted how SV can only play a restricted educational role without teacher supervision due to the students' tendency to reinterpret what the SV is presenting to fit their own misunderstandings, though this affords a new opportunity as to how SV can be used as an effective communication tool.

References

Bergantz D. and Hassell, J. (1991). Information Relationships in PROLOG programs: how do programmers comprehend functionality? *International Journal of Man-Machine Studies*, **35**, 313-328.

Brayshaw, M. and Eisenstadt, M. (1991). A Practical Tracer for Prolog. *International Journal of Man-Machine Studies*, 42:597-631.

Byrd, L. (1980). Understanding the control flow of Prolog programs. In S. Tarnlund (Ed.), *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary.

Coombs, M. J. and Stell, J. G. (1985). A model for debugging Prolog by symbolic execution: the separation of specification and procedure. *Research Report MMIGR137*. Department of Computer Science, University of Strathclyde.

Dawson, R. J., Newsham, R. W., and Kerridge, R. S. (1992). Introducing new Software Engineering Graduates to the 'real world' at the GPT Company. Software Engineering Journal, **7** (3), 171-176.

Domingue, J. and Mulholland, P. (1997). Fostering debugging communities over the World Wide Web. *Communications of the ACM*, **40** (4), 65-71.

Domingue, J. and Mulholland, P. (to appear). Staging Software Visualizations on the Web. *Visual Languages '97.*

Eisenstadt, M. (1984). A Powerful Prolog Trace Package. *Proceedings of the 6th European Conference on Artificial Intelligence.* Pisa, Italy.

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., Price, B. A., and Domingue, J. (1992). Software Visualization: Rederessing ITS Fallacies. In *Proceedings of NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming*, Genova, Italy.

Ericsson, K, A. and Simon, H. A. (1984). *Protocol analysis: Verbal reports as data.* Cambridge, MA: MIT.

Fung, P., Brayshaw, M., du Boulay, B., and Elsom-Cook, M. (1990). Towards a taxonomy of novices' misconceptions of the Prolog interpreter. *Instructional Science*, **19** (4/5), 311-336.

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

Mulholland, P. (1995). *A framework for describing and evaluating Software Visualization Systems: A case-study in Prolog.* PhD Thesis, Knowledge Media Institute, Open University.

Mulholland, P. (to appear). Using a Fine-Grained Comparative Evaluation Technique to Understand and Design Software Visualization Tools. *Empirical Studies of Programmers 7th Workshop.*

Mulholland, P. and Eisenstadt, M. (in press). Using Software to Teach Computer Programming: Past, Present and Future. In M. Brown, J. Domingue, B. Price and J. Stasko (Eds.), *Software Visualization: Programming as a multi-media experience*. Cambridge, MA: MIT

Price, B. A., Small, I. S., and Baecker, R. M. (1991). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, **4** (3), 211-266. Cambridge, MA: MIT

Taylor, J. A. (1988). *PROGRAMMING IN PROLOG: An In-Depth Study of the Problems for Beginners Learning to Program in Prolog.* Unpublished PhD Thesis, Department of Cognitive and Computing Sciences, University of Sussex.

Taylor, C., du Boulay, B., and Patel, M. (1991). *Outline proposal for a Prolog 'Textual Tree Tracer' (TTT).* CSRP No. 177, Department of Cognitive and Computing Sciences, University of Sussex.