The Knowledge Engineering Review

Learning to use the S.1 knowledge engineering tool

R.D. WARD and D. SLEEMAN

Department of Computing Science, University of Aberdeen, King's College, Aberdeen, AB9 2UB, Scotland

Abstract

It is often claimed that it is easy to write expert systems. This claim was examined by monitoring experienced programmers learning to use the S.1 knowledge engineering tool. Their achievements and difficulties were examined using a framework that has emerged from previous research into novices learning to use standard programming languages. Even though the experienced programmers all had several years' experience of programming in more than one standard language, there were similarities between their difficulties in learning to use S.1 and the difficulties of complete novices learning to program in standard languages.

The experienced programmers were however able to overcome their initial difficulties fairly quickly, but it is argued that complete novices would not find it so easy to do so. Also the experienced programmers *did* take time to develop a repertoire of schemata for representing different kinds of factual, judgmental and procedural knowledge. It was concluded that in S.1, as with other programming languages and software tools, it is easy to learn how to do simple things, but difficult, even for experienced programmers to learn how to do more complex things.

No criticism of S.1 is implied. S.1 was found to be a suitable vehicle for introducing non-trivial knowledge engineering concepts, and we believe that similar difficulties would occur in learning to use other knowledge engineering tools.

1 Introduction

It is frequently claimed, especially in the general computing press, that expert system shells make expert systems easy to build. An extreme form of the claim by Dale (1985) was that a particular tool:

"... has taken the concepts of EMycin, and with further tools added in the form of a development engine and other utilities, it has become entirely practicable for a user with no previous programming experience to build a complex expert system and to validate it."

D'Agapeyeff (1985) was more restrained in his analogy with writing programs in Basic:

"... anyone can write a program in Basic if they are either sufficiently motivated or have been given some training. They will not write, at first, a very good program and they should not tackle an ambitious application. Expert systems are only another kind of program. Given an incentive and a little training, anyone in an office can represent some of their know-how ..."

These claims and others like them imply that novices are capable of developing expert systems without too much trouble. But studies of novice programmers completely new to computing, using general purpose programming languages such as Pascal, Basic, Lisp and Prolog, imply that programming is not trivial and that many difficulties are independent of specific programming languages (e.g. Shneiderman, 1980; Sleeman and Soloway, 1986; Sleeman, et al., 1986; Putnam et al., 1986; du Boulay and Sothcott, 1987). This suggests that complete novices could be expected to have difficulties with knowledge engineering software. Indeed, some difficulties might also be encountered by experienced programmers using a knowledge engineering tool for the first time as

this would still entail working with unfamiliar concepts, unfamiliar syntax, and an unfamiliar software environment. Why should knowledge engineering tools be different from other software?

The claim that expert system shells make expert systems easy to build was examined through informal discussions with, and observations of, experienced programmers (largely, third and fourth year computing science undergraduates) learning to use S.1 version 2.1, running under Unix on a Sun 3/160, as their first knowledge engineering tool. In this version of S.1, interaction between the user and S.1 takes place in text alone mode. In other versions, S.1 provides graphics orientated tracing and debugging tools, but these were not used in this study.

The subjects were eight students who used S.1 for approximately 8 hours each as part of an A.I. course, three students who used S.1 extensively over two terms for their final year projects, one student who spent a week, full-time, translating into S.1 an existing knowledge base that interprets the entrance requirements for Computing Science courses, and one research fellow (the first author). No subject had less than two years programming experience in at least two languages, mainly Pascal and POP-11.

Subjects were pedagogically well-supported. Most were introduced to S.1 through specially written teaching notes which describe, with examples, how to construct simple S.1 knowledge bases (Ward, 1987). Section 2 below, is based upon these notes. (The documentation supplied with the system contained few examples and the example knowledge bases provided were found to be too sophisticated for beginners to understand). Most were also attending a course of lectures and practical, supervised sessions in A.I. and Expert Systems, in which one lecture (by the second author) presented an overview of S.1.

Subjects' difficulties were categorized within a framework that has emerged from previous research into novices learning to use standard programming languages. Du Boulay and Sothcott (1987) suggest that knowledge of programming can be considered under four headings:

- Orientation
- Knowledge of the syntax and semantics of a language
- Pragmatic knowledge of how to conduct an effective dialogue with the programming environment and
- Tactical and strategic knowledge about decomposing tasks and composing programs.

We believe this framework provides a principled approach for evaluating the usability of software tools.

Additionally, we have hypothesized that the causes of students' programming difficulties are:

1a Logical inconsistencies and logical complexity in the language contructs;
1b poor or uninformative error messages (both syntactic and run-time);
1c inadequate teaching methods or materials (i.e. the language constructs and error messages are comprehensible, but the teaching methods and materials fail to convey this).
2 Student-generated misunderstandings which arise because of the background knowledge or lack of it, which students bring to bear (i.e. the concepts involved are consistent and not complex, and the teaching methods and materials are of good quality, and appropriate).

Thus, errors of class 1 result primarily from factors outside the students' control, whereas class 2 errors are student-based*. Also, errors of class 1 will tend to vary according to the programming language, whereas class 2 errors are likely to be common across several languages, especially where languages involve common concepts.

The importance of assigning provisional causes to errors is that some sources of error can then be removed. For example, errors which arise because of inconsistencies in the language (i.e. errors of class 1a) might be remedied by redesigning the language. Errors of class 1b suggest that alterations to the implementation of the system are required, and also possibly a redesign of constructs. Errors of class 1c require further work on instructional materials and methods, and documentation.

* This overall framework may also be applicable to other subject areas.

Finally, errors of class 2 are extremely difficult to remedy because students bring widely varied and often ill-defined background knowledge to bear on their learning task.

2 The S.1 knowledge engineering tool

S.1 is a knowledge engineering tool developed by Teknowledge, suited to diagnosis and classification problems. Its origins are in MYCIN (Shortliffe 1976), so it therefore uses mainly a rule-based representation, with a backward chaining control scheme. It also supports frame-based and procedural representations of knowledge; through these other control schemes are possible,

Figure 1 A rudimentary S.1 knowledge base

```
define control.block   identify.tree
   ::invocation        top.level
   ::body
      begin vars t:tree;
      create.instance tree called t;
      determine treename[t];
      display new.line() ! indent() !
            "I think your tree is" ! treename[t]
      end
   ::translation "identify the tree"
end.define

define class tree
   ::number.instances   1
   ::full.instance.translation "your tree"
end.define

define attribute treename
   ::defined.on              tree
   ::type                    text
   ::legal.values            {oak, beech, elm, silver.birch}
   ::determination.means     {try.rules}
   ::legal.means             {try.rules}
   ::translation "the name of " ! instance.trans(tree)
end.define

define attribute barktexture
   ::defined.on              tree
   ::type                    text
   ::legal.values            {smooth, fissured}
   ::determination.means     {query.user}
   ::legal.means             {query.user}
   ::prompt                  "What is the texture of the bark ?"
   ::translation "the texture of the bark of "
                 ! instance.trans(tree)
end.define

define rule bark1
   ::applied.to   tree
   ::premise      barktexture[tree] is smooth
   ::conclusion   treename[tree] = beech <0.2>
end.define

define rule bark2
   ::applied.to   tree
   ::premise      barktexture[tree] is fissured
   ::conclusion   treename[tree] = oak <0.2>
end.define
```

Figure 2 Two consultations of the knowledge base from Figure 1, showing a full event history

```
                S.1
Copyright © 1986 by Framentec SA
            All rights reserved.
Welcome to S.1 Version 2.1 (R19) [2D0000]

→ load simple.kb quiet
Knowledgebase is loaded and consistent.
→ trace all
→ start
This will be case number 1
0:1   Control Block identify.tree invoked with no arguments.
      (It was invoked as the TopLevel Control Block).
0:2   Attempt made to create an instance of tree.
      (This is caused by Control Block identify.tree with no arguments)
0:3   Tree #1 created.
0:5   Determination started for treename of tree #1.
      (Determination caused by Control Block identify.tree with no
      arguments).
0:6   Seeking to determine treename of tree #1 by rules.
      (Seeking caused by determination of treename of tree #1).
0:7   Rule bark1 applied for treename of tree #1.
0:8   Determination started for barktexture of tree #1.
      (Determination caused by Rule bark1 applied to tree #).
0:9   Seeking to determine barktexture of tree #1 by asking.
      (Seeking caused by determination of barktexture of tree #1).

1:    What is the texture of the bark ?
1> smooth
1:1   Barktexture of tree #1 concluded to be smooth <1.0> (caused by
      user input).
1:2   Barktexture of tree #1 concluded to be fissured <-1.0> (caused
      by propagation).
1:3   Barktexture of tree #1 is determined.
1:6   Rule bark1 succeeded for tree #1.
1:7   Treename of tree #1 concluded to be beech <.2> (caused by bark1).
1:8   Rule bark2 failed by premise preview for tree #1.
1:10  Treename of tree #1 is determined.

  I think your tree is beech <.2>
1:12  Control Block identify.tree with no arguments exited.

→ start
This will be case number 2
(Identical trace to case number 1, 0:1 to 0:9 removed)
1:  What is the texture of the bark ?
1> unknown
1:2   Barktexture of tree #1 is determined.
1:4   Rule bark1 failed for tree #1.
1:5   Rule bark2 failed by premise preview for tree #1.
1:7   Treename of tree #1 is determined.

  I think your tree is unknown
1:9   Control Block identify.tree with no arguments exited.
```

including a limited form of forward reasoning. S.1 also possesses certainty handling mechanisms, based upon certainty factors within the range $<-1.0>$ to $<1.0>$ inclusive.

S.1 knowledge bases are collections of different kinds of objects: control blocks, classes, attributes, rules, class types, value hierarchies, functions, external functions and external control blocks. The first four of these will appear in nearly every S.1 knowledge base. The others may or

may not be used. Each object is defined in a block of text, each block consisting of several slots. Each type of object has its own essential slots, plus optional slots dependant upon the presence of other objects in the knowledge base. The whole knowledge base must conform to S.1's syntax. Figure 1 shows a rudimentary S.1 knowledge base. This represents the beginnings of an expert system to help the user to identify the names of trees. The example, which has just one control block, one class, two attributes and two rules, contains knowledge about the bark texture of just two types of tree. A real-life application would clearly contain more knowledge than this.

It can be seen from figure 1 that rules in S.1 have the form "if PREMISE then CONCLUSION". The rules are used by the in-built backward-chaining mechanism in reasoning about the class-attribute-value triples defined by the class and attribute definition blocks. Backward-chaining is initiated by the "determine" statement in the control block. Figure 2 shows a full trace of the knowledge base in figure 1 being loaded and consulted. Normally, only the system's questions and the user's responses would appear on the screen, not the full trace. S.1 also possesses inbuilt explanation facilities which use text from the "::translation" slots in order to answer "what", "why" and "how" questions during a consultation.

S.1 is not as complex as some knowledge engineering tools, for example KEE (Knowledge engineering environment) from Intellicorp, Knowledge Craft from Carnegie Group and ART (Automatic Reasoning Tool) from Inference Corporation. On the other hand it allows the user more scope than the simpler expert systems shells available on personal computers.

3 Difficulties of orientation

Amongst the first problems faced by novice programmers with no previous experience of computers whatsoever, are the problems of orientation to programming and to the different ways in which the terminal can operate (du Boulay, 1986). New students are often unsure about what programming is and what types of problem can be tackled. They experience difficulties at all sorts of levels: they may fail to realise that the text appearing on a terminal shows a historical record of interactions between user and computer, or that in order to type an exclamation mark it is necessary to hold down "shift" whilst pressing the "1" key.

Even experienced users of computers attempting to use unfamiliar software for the first time may be subject to problems of orientation and conceptualization. At the very least, any new software environment will cause difficulties because of its unfamiliarity. But for those new to expert systems, it may not be clear what expert systems are and what kinds of problem they can tackle. Thus one of the students observed, constructed an expert system to identify different whiskies. As this required the person consulting the knowledge base to recognize and discriminate subtle differences in taste—skills which only an expert whisky taster could be expected to possess, it was an inappropriate application of the technology.

Furthermore, for experienced programmers prior knowledge of computing may get in the way of developing an understanding of expert systems. For example, concepts associated with variables and their assignment, as they relate to standard programming languages, are not always appropriate in S.1. This point is expanded in section 6.5.

Problems of orientation clearly belong to the second class of difficulties described in section 1: they primarily result from weakness in students' understanding. However, given adequate support, students should not find these problems insurmountable.

4 Difficulties with syntax and semantics

The syntax and semantics of the statements used in a knowledge engineering tool must be learned, just as for any programming language or software tool. Common mastery of statements, i.e. being able to correctly produce them and know their meaning, is usually regarded as the easiest level of

knowledge to acquire (e.g. Kinzer et al., 1985). Nevertheless, experienced programmers were observed to make frequent syntactic slips in their early use of S.1. Typically, these were misspellings, and the inadvertent omissions of semi-colons, exclamation marks and the "end.define" block terminator. Inevitably, a few slips like these will occur in a verbose language like S.1. Most subjects were irritated by the necessity of having to type entities such as "new.line()", "full.instance.translation" or "{query.user}" in full.

Syntactic problems of novices learning standard languages include over-generalisations and interaction effects between syntax and layout (Shneiderman, 1980). These problems were also observed in proficient programmers using S.1 for the first time. Typically the use of the full stop, common in the many S.1 statements that consist of more than one word (e.g. "query.user", "end.define"), was over-generalized to items in which it should not appear, for example "multivalued", "subvalues", "define rule" and "trace all". There seems to be an intuitive inconsistency between the correct usage of the full stops in "define <object_name>" and "end.define". Substitution of a full stop for the colon in a variable declaration such as "t:tree", the omission of full stops in identifiers such as "defined.on", and the use of round instead of pointed brackets for certainty factors were also observed.

Interaction effects with layout occurred too. For example if a layout was adopted in which the "end.define" of each block was indented to the level of the preceding block slots, then there was a tendency to precede "end.define" with the double colon that appears at the beginning of each block slot.

Problems of syntactic over-generalization, and interaction effects with layout, appear mainly to result from inconsistencies within the S.1 language; i.e. they fall into the first class of difficulties we have described.

There were also difficulties that appeared to involve a semantic element. These appeared to be of class 2. Examples are (i) lack of appreciation of the importance of instantiation, leading to the omission of instance variables after an attribute name; (ii) confusion between a class name and its instance variable and (iii) confusion between the "::determination.block" and "::determination.means" slots in attribute definition blocks.

5 Difficulties with pragmatic knowledge

Novices completely new to computing must develop what du Boulay and Sothcott (1987) call their knowledge of how to "do programming". This covers the ability to follow and adapt to changes of environment between monitor, editor, compiler and run-time system, and also the ability to understand system error messages.

Experienced programmers should be well practised in switching between software environments, but they did have problems in navigating between S.1's four different modes of operation: knowledge-base preparation, knowledge-base consultation, consultation-break and loading-break. For example, it is not possible to exit S.1 in mid-consultation: one must first enter the consultation-break mode by means of a <CONTROL-C>, then return to the preparation mode by means of the "abort" command, and then leave S.1 using the command "exit". This is clearly a class 1 difficulty—a weakness in the software environment.

The syntax and run-time error messages produced by S.1 appeared to be no more helpful than those produced by standard Pascal compilers and just as misleading. Messages were inaccurate and frequently referred to lines other than where the error had been made. Messages also varied oddly, according to the context of the error. For example the omission of the instance variable "[tree]" from the end of an attribute reference in a rule premise produced the message "expected <Rule Premise> missing", whereas the same omission in a rule conclusion was described as "Attribute Reference mismatched Instance".

One error that was particularly difficult to understand occurred with the S.1 object known as a value hierarchy. Value hierarchies allow all the legal values that may be assumed by an attribute to

be defined in one place, rather like the "type" declaration in Pascal. In the knowledge base of figure 1, the set of legal tree names could be extended and defined in a value hierarchy as follows:

```
define value.hierarchy treenames
   ::subvalues {oak, beech, elm, silver.birch, rowan,
                sycamore, lime, birch, pinaceae.family}
   ::translation "the names of the trees"
end.define
```

And the "::legal.values" slot of attribute "treename" would then become "::legal.values treenames". But if in error, the entry in this slot was enclosed in curly brackets as if it were a value rather than a value.hierarchy, i.e. "::legal.values {treenames}", the knowledge base could still be loaded and consulted normally until an attempt to determine a value for the attribute occurred. In the present example this would have occurred when, and only when, rule bark2 fired:

```
1:  What is the texture of the bark ?
1 > fissured
Error
6 : oak is an illegal value for attribute treename
Aborting consultation.
```

Again these difficulties appear to be of class 1, caused mainly by software weaknesses. Despite the inconsistencies, the experienced programmers of this study, usually overcame their errors unaided in the end. It seems doubtful that complete novices would be able to do so.

Subjects tended to develop their knowledge bases incrementally, gradually increasing the amount of knowledge represented by adding more and more attributes and rules. Building a knowledge base therefore required a methodology for testing each new section once it had been added and syntax errors removed. One technique involved adding attributes one at a time together with a temporary control block to determine and display the value of the new attribute. Rules were also added a few at a time and methodically tested in order to ensure that the new rules alone had the effect intended and also that there were no unforseen interactions with existing rules. Experienced programmers were able to carry out this methodology competently. For complete novices this would surely add yet another level of complexity to their learning task and lead to difficulties of class 2.

6 Difficulties with tactical and strategic knowledge

Du Boulay and Sothcott (1987) suggest that with standard programming languages there are two complementary difficulties—decomposing a problem into manageable sub-problems, and then choosing and linking together the most appropriate program constructs to represent the sub-problems. There is evidence to suggest that experienced programmers are able to perform these tasks by drawing on and joining together standard chunks of code, or "idioms", or "programming cliches", or "schema" with known effects (e.g. Ehrlich and Soloway, 1984).

Working from the example shown in figure 1 and its associated teaching notes, all of the experienced programmers that were observed using S.1 were quickly able to construct knowledge bases consisting of several attributes defined on one class. These knowledge bases contained up to 20 or 30 rules to determine the value of one attribute from the values of the other attributes, most of which were determined by querying the user. The top level control block usually consisted of a "create.instance" statement, followed by a single "determine" statement, followed by a "display" statement. In rule-based expert systems this is probably the simplest schema. Applications included umpire's decision making in cricket, the diagnosis of football team weaknesses and the identification of fish, dogs and, as already mentioned, varieties of whisky.

Subjects were also soon able to introduce subsidiary control blocks to be called by the "top.level" block with an "invoke" statement for the purposes of printing headings and displaying results.

Other easily understood concepts were Boolean attributes and their associated rule premises, negative premises and simple set membership, as shown in the following rule premises:

```
::premise   attribute[instance]
::premise   not attribute[instance]
::premise   attribute[instance] is.not value
::premise   attribute[instance] is in {value1, value2, ..., valueN}
::premise   attribute[instance] is.not in {value1, value2, ..., valueN}
```

At this level there were few Class II difficulties.

However as their knowledge bases became larger, tactical and strategic difficulties became common. Subjects met with problems requiring deep and careful thought. We believe these problems to be of class 2 in that they involve new concepts which take time and effort to master. The following issues were common:

6.1 What knowledge should be represented by classes, and what should be represented by attributes?

The answer is not always clear. If for example the "tree" knowledge base of figure 1 were to be extended to include knowledge about the shapes of leaves, should a new attribute "leaf.shape" be defined on the class "tree", or should a new class "leaf" be defined, with "shape" as one of its attributes? This area of difficulty became more pronounced with attempts to introduce more structure into the knowledge, which in S.1 may be achieved through the use of concepts such as attribute subsumption, hierarchical values and class association.

6.2 How should rule premises and rule conclusions be organized, and how should rules inter-relate?

The rule premises in figure 1 contain just one condition, and the rule conclusions just one statement. S.1, in common with other shells, permits rules with compound premises and conclusions. For example:

```
::premise     attribute1[instance] thought.not value1 and
                 (attribute2[instance] or attribute2[instance])

::conclusion  begin
              attribute4[instance]=value1 <-0.2>, value2 <0.4>
              attribute5[instance]=value1 <1.0>, value3 <-1.0>
              end
```

This means that the same knowledge can be represented by rules in different ways—for example in a series of rules with simple premises and conclusions or in a single rule with a compound premise and a compound conclusion. If simple premises are used, the degree of certainty about a particular attribute value can accumulate incrementally as each piece of evidence causes successive rules to fire. Alternatively this knowledge can be combined into compound premises, but it must be remembered that in S.1, if any conjunction of a compound premise is false, then the whole rule will fail to fire.

It is not always easy to decide which approach is most suited to a particular application, but if insufficient care is taken at this stage then it can be difficult to detect later from the operation of the system alone that the rules do not represent the domain knowledge in the intended way. Premises and conclusions can become difficult to understand, and the example rule given by Lan, Panos and Balban, (1988), illustrates the complexity that can be involved: note here the restricted instance variables declared in the "already.existing" and "for.all.existing" expressions. This degree of complexity can lead to possible omissions or duplications of rule-based knowledge. If knowledge is omitted then certain pieces of evidence will play no part in the system's reasoning process. Duplicated knowledge will cause evidence to be given double weight.

The construction of rule premises and conclusions therefore demands a systematic approach,

with care taken to ensure that all eventualities have been taken into account as each new group of rules is added. One strategy that subjects used here was to make each new rule simple, and to look for possible optimizations later. Another strategy involved the construction of N by N contingency tables to cover different sub-units of knowledge. This latter technique became more and more prone to error as the number of attributes and values increased.

6.3 How should attributes values be organized?

S.1 allows the knowledge engineer to organize attribute values hierarchically. In the following example a single rule is used to preclude all trees with broad leaves from being identified as any member of the pinaceae family:

```
define value.hierarchy pinaceae.family
    ::subvalues {pines, larches, firs, spruces, hemlocks}
end.define

define value.hierarchy pines
    ::subvalues   {scots.pine, maritime.pine, stone.pine,
                   shore.pine, corsican.pine, alleppo.pine,
                   swiss.stone.pine, monterey.pine}
end.define

define rule leaves20
    ::applied.to  tree
    ::premise     leaf.shape[tree] is broad.leaved
    ::conclusion  treename[tree]=pinaceae.family <-1.0>
end.define
```

Hierarchical organization of values can lead to simpler rule premises and conclusions, but arriving at the "right" organization is often difficult and several false starts may be unavoidable.

6.4 What levels should be used for certainty factors in rule conclusions?

Each of the possible values for each of the attributes defined on each instance of a class is initially unknown, i.e. it has a zero certainty factor. During a consultation these factors may change, for example, because of the effects of a rule conclusion. If two or more rules produce conclusions for the same attribute value then their factors will be combined. However factors of $<1.0>$ or $<-1.0>$ represent certainty and once certainty has been concluded for the value of any attribute then the attribute will be marked as having been determined, and no further evidence will be considered.

Certainty factor values are arbitrary, and it is important to recognize that they are not exact probabilities. A rule of thumb devised by one subject was to select certainty factors so that the final highest values of attribute fall within the range $<0.5>$ to $<1.0>$. In order to achieve this it is necessary to have some idea about how many combinations are likely to be made during a consultation, and also to know about the ways in which combinations of certainty factors are calculated. For example three pieces of evidence, each with a certainty factor of $<0.3>$ will combine in S.1 to produce a factor of $<0.657>$. Therefore a further single piece of evidence that is judged to have the same value as these three together should be given a value of $<0.657>$. The "balancing" of certainty factors is a delicate matter.

6.6 Attribute values and their determination

The fact that the values of an attribute cannot be altered once determined, is central in S.1 and other predicate calculus based systems. In S.1, attributes are marked as determined either when there

remain no more untried rules that apply to that attribute, or when a "definite" certainty factor (i.e. <1.0> or <−1.0>) has been concluded.

Another key concept is that attributes values are, in fact, sets of paired values and certainty factors and that therefore an attribute may hold several values simultaneously with varying certainty factors.

Attributes are thus very different from variables in procedural programming languages, yet many subjects attempted to treat them as variables at some point by assigning values directly through control block assignment statements, which is illegal in S.1. However, novices without prior experience of computing might not be as susceptible as experienced programmers to this misconception.

6.6 S.1 data types

Everyone had difficulties in understanding S.1 data types. For example, in addition to the more usual text, integer, real and boolean types, S.1 also possesses among others, types for certainty factors (type cf), pairs of values and certainty factors (type value.cf), and sets of pairs of values and certainty factors (type value.cf.set). It is easy to confuse these different types and the inbuilt functions provided to manipulate them. This can lead to difficulties when attribute values are determined by user-defined functions, as opposed to being determined by the more straightforward "query.user" and "try.rules" means. The knowledge engineer must be clear about whether a function concludes an individual value.cf or a full value.cf.set and whether the result returned by the function is for combination into an existing value.cf.set of attribute values or whether it fully determines the values of the attribute.

6.7 More advanced control structures

Many subjects also had difficulties in understanding the concepts required to construct knowledge bases that went beyond the simple "create.instance-determine-display" schema.

For example, one technique for representing time, or for allowing the user to change selections or decisions made during the course of a consultation, is the recursive creation of successive instances of the same class. One way in which this can be implemented in S.1 is by a post-determination control block, that is a control block invoked immediately after a particular attribute value has been marked as determined. The following control block would be invoked immediately after the statement "determine treename[t]" had finished execution:

```
define control.block alter.tree
    ::invocation   post.determination
    ::arguments    treename:attribute, t:tree
    ::body
        begin
        . . .
        <statements>
        . . .
        if <condition> then determine treename[t]
        end
end.define
```

Provided that a "::post.determination.block" slot with the entry "alter.tree" was included in the definition block for the attribute "treename", then successive instances of the class "tree" would be recursively created until a false tail end condition was encountered in the "alter-tree" control block.

The generate and test control structure used by Lan, Panos and Balban (1988), shows a still higher degree of sophistication.

7 Conclusions

Experienced programmers were able to learn to construct simple knowledge bases with S.1 fairly rapidly. They were able to cope with their inevitable errors of syntax and semantics, and to effectively debug and extend their knowledge bases. It can therefore be said that learning to construct simple knowledge bases with S.1 is perhaps no more difficult than learning to program in Pascal. Conversely, we do not believe it to be easier.

Although experienced programmers were usually able to overcome their structural and pragmatic difficulties fairly easily, it seems very unlikely that this would be the case for novices completely new to computing. The kinds of syntactic, semantic and pragmatic difficulties that arose, were similar in nature to those found in other studies of novice programmers. These results suggest that complete novices would therefore be likely to have just as many problems with S.1 as they would with Pascal. Thus the claims made by Dale (1985) and d'Agapeyeff (1985) and others, that anyone in an office with no previous programming experience can build expert systems without difficulty appear to be quite unjustified.

We maintain this even though most syntactic, semantic and pragmatic difficulties belonged to the first class of difficulties we described in section 1, resulting primarily from weaknesses in S.1, and being therefore specific to S.1. We believe similar problems are likely to occur in other knowledge engineering tools because (i) expert systems shells have complex control flow which can give rise to unpredictable errors and omissions and (ii) it is well known that there is a difference between the point at which a parse fails and the error which causes the failure, and that error messages usually report the former (du Boulay and Matthew, 1984).

Perhaps we have been a little unfair in deprecating Dale and d'Agapeyeff, who were after all, referring to small expert systems shells for personal computers, which are not as elaborate as S.1. Also, to some extent, several shells for small computers do protect the user from syntactic, semantic and pragmatic difficulties through prompts and menus. Nevertheless, the observations reported here were rarely concerned with the more difficult S.1 concepts, such as successive instantiation, or attributes and rules defined across several classes through the use of class types. Many of the observations refer to early experiences with S.1 when most subjects were constructing the kinds of knowledge base that might have equally well been implemented with smaller shells.

Furthermore, there still remains the area of tactics and strategy. This contained a number of profound conceptual difficulties for subjects and these were difficulties of class 2, likely to occur in other rule-based expert system shells. In other words, once a knowledge engineer wishes to go beyond the simplest knowledge base structures, for example by organizing factual knowledge into groups or hierarchies; by "tuning" judgmental knowledge through alterations to rule order or the values of certainty factors; by altering the control flow of the system; or by using successive instantiation, then difficulties in assembling the components of knowledge bases become more likely. The relevant schema need to be learned, just as in any other programming language.

Despite subjects' difficulties, this paper is not a denigration of the S.1 tool. S.1 was considered by most subjects to be a suitable system for becoming familiar with non-trivial knowledge engineering concepts.

Acknowledgements

The authors would like to thank the anonymous referee, Dr John Fox and Ian Kirby for their constructive criticisms of earlier drafts of this paper.

References

d'Agapeyeff, A, 1985. Interviewed in *Computing the Magazine*, 20 June 1985, p. 31.

du Boulay, B, 1986. "Some difficulties of learning to program" *Journal of Educational Computing Research* **2**(1) pp. 57–73.

du Boulay, B and Matthew, I, 1984. "Fatal error in pass zero: how not to confuse novices" *Behaviour and Information Technology* 3 pp. 109–118.

du Boulay, B and Sothcott, C, 1987. "Computers teaching programming: an introductory survey of the field" In: *Artificial Intelligence and Education: Learning environments and intelligent tutoring systems*, Lawler, R W and Yazdani, M, Eds, Norwood, New Jersey: Ablex, Ch. 16.

Dale, F, 1985. "Getting to grips with an expert" *Computer Systems*, December 1985 pp. 31–35.

Ehrlich, K and Soloway, E, 1984. "An empirical investigation of the tacit plan knowledge in programming" In: *Human Factors in Computer Systems*, Thomas, J C and Schneider, M L, Eds, Norwood, New Jersey: Ablex, pp. 113–133.

Kinzer, C, Littlefield, J, Delclos, V R and Bransford, J D, 1985. "Different LOGO learning environments and mastery: relationships between engagement and learning" *Computers in the Schools* pp. 33–43.

Lan, M S, Panos, R R and Balban, M S, 1988. "The Press Lineup Advisor: An Expert System for Newspaper Printing Press Configuration" *The Knowledge Engineering Review* (this issue).

Putman, R T, Sleeman, D, Baxter, J A and Kuspa, L K, 1986. "A summary of misconceptions of high school basic programmers" *Journal of Educational Computing Research* **2**(4) pp. 459–472.

Shneiderman, B, 1980. *Software psychology*, Cambridge, Massachusetts. Winthrop Publishers.

Shortliffe, E H, 1976. *Computer-based medical consultations: MYCIN*, New York: Elsevier.

Sleeman, D, Putnam, R T, Baxter, J A and Kuspa, L K, 1986. "Pascal and high school students: a study of errors" *Journal of Educational Computing Research*, **2**(1) pp. 5–23.

Sleeman, D and Soloway, E, (Eds.), 1986. "Special issue on Novice Programmers" *Journal of Educational Computing Research* **2**(1).

Ward, R D, 1987. *Getting Started with the S.1 Knowledge Engineering Tool*, Documentation AUCS/D8702, Department of Computing Science, University of Aberdeen.