A graphical inference mechanism

Abstract: *This paper describes an inference system which lends itself to graphical representation. An implementation of the system is described, and its application in a legislation based domain is illustrated. The methodology for knowledge elicitation which the system is intended to support is briefly indicated. The algorithm is described, and semantics for the system are given.*

PETER MOTT
SIMON BROOKE

Department of Systems
University of Lancaster
Gillow House
Lancaster
LA1 4YX
England

Introduction

The paper is divided into four sections. In the first section, we describe the background to our work: the domain in which our inference engine was intended to function, and the issues raised by the special problems of that domain. In the second section, we describe in general terms the prototype inference engine we designed to meet these problems, concentrating particularly on the graphical rule interface, and the explanation generator. We have concentrated the technical material in the third section, where we present the algorithm for the inference process, and a semantics for the logic used. In the final section we discuss issues relating to the representation of legislation.

1. Background

The Department of Health and Social Security (DHSS) is charged, among other things, with administering the range of welfare benefits established by legislation and with advising Ministers on possible changes and additions thereto. The Alvey-DHSS Large Demonstrator project aims to explore the potential of Intelligent Knowledge Based Systems for assisting the Department in these tasks. In the present paper we focus on just one area of potential support: the process of adjudicating benefit claims as carried out in Local Offices of the DHSS by Adjudication Officers.

We describe in detail a system (which we call DTrees) that implements a novel way of structuring legal rules. Instead of a simple IF..THEN... structuring of rules this system employs an IF..THEN..UNLESS.. format. Indeed, the 'unless' begins to predominate until we see the structure in a system of legal rules as exactly a structure of common cases and exceptions proceeding through several levels from the ordinary to the recondite.

The ideas presented here were developed within the Large Demonstrator project as one of several concurrent avenues of investigation into possible support for Adjudication Officers within the DHSS. It is not possible within the scope of this paper to discuss in any great detail the operation of the Local Office system of the DHSS. However, some discussion is necessary both to describe the problems to which the DTree system offers (at least partial) solutions and to forestall any impression that we claim more for the system than we actually do.

Basically, the context in which Local Office decision making occurs is one which involves the application of a large and interrelated body of rules of varying complexity and difficulty and, indeed, accessibility. Moreover, the Local Office staff who must determine the effect of these rules, themselves occupy widely varying roles and exhibit considerably different degrees of expertise. As a result the Local Office context is one in which both the rules which have to be applied and the roles of those who have to apply them can be ambiguous and even contradictory. Additionally, information coming into the system from claimants is often diverse both in structure and type. Therefore any interaction between client and Local Office will involve problems of definition and explanation. The matching of relevant rules to particular cases is by no means as simple as the codified nature of the rules might suggest [1].

We are aware of these difficulties and do not suggest that the present system removes them. Furthermore, any final system of support would have to include components hardly resembling a rule driven system at all. Indeed possibilities for such support are being actively considered in the Project, for example [2], but we shall not consider them further here. On the other hand it does not do to overstate them. We think that a large proportion of cases are straightforward enough or, even if this be disputed, a large proportion will be treated as straightforward. That is inevitable given the volume of cases and the time and staff available to decide them. There is thus a place for a rule driven system to render as speedy and efficient as possible the bulk of routine cases and thereby free time for the consideration of difficult ones.

Adjudication Officers must decide claimants' eligibility over a wide range of welfare benefits. They do not work as such all the time. In fact only about 20% of their working day is spent in adjudication. Furthermore, there is a very large volume of work to be done, so they work under considerable pressure.

The Adjudication Process is independent of the Secretary of State and has its own levels of authority culminating in the Chief Adjudication Officer. He is supported by the Office of the Chief Adjudication Officer (O.C.A.O.) which has a general supervisory role over the adjudication process. It monitors the quality of the decision making of the Adjudication Officers, issues written guidelines to assist Adjudication Officers in their decision making, and distributes information about changes in the benefit rules. This last point needs emphasis.

The rules concerning welfare benefits are frequently modified, both to correct local anomalies and of course to reflect more substantial political changes in the provision of welfare. A static support system then, constructed at considerable expense around a fixed set of rules, cannot be useful because it will become outdated almost immediately. What is required is a system that can be modified and updated easily by the officials in the O.C.A.O.

The general context is then of a (very) large organisation in which relatively numerous junior officials apply extensive and frequently changed legislation in limited time under the supervision of relatively few senior officials. Fairly obviously, the task of an IKBS support system is to maintain and if possible improve the decision making quality of the former and render more effective the supervision exercised by the latter. The present system is designed to contribute a component to that goal.

We have aimed to design a system which would access the legislative rule base (however extensive it may be), which would propose a decision for the case in question and then draft a letter explaining to the claimant that decision. If the case was straightforward the Adjudication Officer would accept the suggestion offered by the machine. Otherwise s/he can study a graphically presented trace of the machine's inference processes to assist him/her in formulating an alternative decision. Traces are notoriously unreadable, but the graphic technique described below renders them, we believe, scrutable and hence useful.

Such a system would relieve Adjudication Officers from the drudgery of hand drafting standard form letters for straightforward cases. Instead the system would do this — using indeed more extensive and informative letters (originally prepared in O.C.A.O.) than is possible given the need to hand draft. The advantages for the claimant are: a better explanation of his/her case and security against 'careless mistakes'; for the O.C.A.O. they are the ability to change and update the system far more quickly, for the Adjudication Officer freedom from drudgery and the time to exercise judgment on the more problematic cases.

2. Using the system

The object of this section is to provide the reader with an informal introduction to the logic of the DTree system, and to the ideas behind it, by describing the construction and use of a small knowledge base. We will briefly describe an experimental implementation in order to make the ideas clear. The more technical aspects will be covered in the next section.

2.1 Arboretum

The implementation, called 'Arboretum', which was written in InterLisp-D [3] using Loops [4] object oriented facilities, is designed to allow people to manipulate DTree structures through graphical representations: to build arbitrarily large knowledge bases, to use these to provide answers to questions about objects in domains admitting incomplete information — and to provide natural language explanations of these answers. The inference process by which answers are produced is shown as an animated graph. The user can ask the system how the value of any particular feature was arrived at, and what that value was.

One of the most significant advantages we would claim for the system is that it is extremely simple for relatively naive users to build knowledge bases. Knowledge engineering with this system requires no understanding of computer language, no very abstruse knowledge about the machine, no complex calculation of weightings and probabilities. The logic is designed to facilitate a methodology which we call 'elicitation by exception'.

2.2 Elicitation by exception

The knowledge engineer's task, using this methodology to build a rule, is simply to ask: is a given predicate normally true? If it were true, would it normally be a sufficient condition? And having got this far, is there anything at all which would overturn the decision? If nothing could, the process terminates; but if there is some more abstruse factor which could still cause a change of mind, then that is added as a new condition and once more one asks if there is anything further that could cause a change of mind. Thus we proceed down a conceptual tree where at each level the decision just made is reversed.

Intuitively the deeper the level, the more unlikely the situations that occupy that level. It is our suggestion that the structure of exceptions that can explicitly be recovered by the knowledge engineer in this way is what is implicitly and imperfectly represented by the certainty factors in classical expert systems.

We believe that the use of a graphical interface also contributes greatly to simplicity of use. It is interesting to note that this approach has similarities to that followed by Richer and Clancey in Guidon-Watch [5]. These similarities will be discussed later, for the moment we just observe that both these graphical interfaces exploit the facilities provided by InterLisp-D on the 1100 series machines, which include a large bit-mapped display, software support for window/icon/mouse user interfaces, and sophisticated tools for building graphs which can be manipulated by the user.

To illustrate the use of the system, let us assume that, as an officer in O.C.A.O., we want to build a rule for 'Entitled to Widow's Allowance'. We will encode from the Social Security Act 1975 [6], chapter 14, section 24, as amended by the Social Security (Miscellaneous Provisions) Act 1977, chapter 5, section 22(2). This reads:

> 24.-(1) A woman who has been widowed shall be entitled to widow's allowance at the weekly rate specified in relation thereto in Schedule 4, Part I, paragraph 5, if:
>
> (a) she was under pensionable age at the time when her late husband died, or he was then not entitled to a Category A retirement pension (section 28); and
>
> (b) her late husband satisfied the contribution requirement for a widow's

allowance specified in Schedule 3, Part 1, paragraph 4.

(2) The period for which widow's allowance shall be payable shall be the 26 weeks next following the husband's death:

Provided that the allowance shall not be payable for any period after the widow's death or remarriage or for any period during which she and a man to whom she is not married are living together as man and wife.

In addition to the legislation itself, the officer would be required to take into account the case law, and previous decisions by the Social Security Commissioners. An example here would be the decision R(G) 2/79 [7], which precludes a claimant from receipt of benefit where the entitlement, based on her status as a widow, is the direct result of an unlawful act of murder or manslaughter even though all the statutory requirements may be satisfied and benefit would otherwise be paid.

It is clear that the above fragment — in common with much legislation — involves a structure of exceptions. We can see immediately that a possible structuring takes the form: entitlement false, unless entitling conditions satisfied, in which case true, unless overriding consideration, in which case false. We can represent this in a DTree.

2.3 Building the rule

The first action is to bring up a display for the feature 'Entitled to Widow's Allowance'. We do this by selecting it from the 'Open Display' menu, or, if it does not already exist, by selecting 'New Feature'. Then we must introduce the feature. As no rule yet exists, the display will show just one (root) node.

To extend it, we point to the root node and select 'Add Node'. We again select 'New Feature' from the features menu that appears, and use the feature inspector which then opens to set up the properties of the next feature, such as its name ('Satisfies conditions for Widow's Allowance'), whether it is normally true in the world and whether it is something the user can reasonably be asked about. This process is repeated for further nodes to achieve a rule like this:

Before going any further, it will be useful to discuss the difference between a 'feature' and a 'node'. A 'feature' is a predicate known to the system; it is something which may be true of an object in the domain. For instance, in the Adjudication Officer's domain, 'widow' is something that can be true of a person. Each feature occurs just once in the system: it is global. As a fundamental building block, it may have properties: the most important of these is that it may have a rule structure, a 'DTree', which may be evaluated in order to establish its value. Another significant property, which all features must have, is a 'default value': the feature must know whether it is usually true, or false, in the domain.

But within a particular DTree we need to be able to refer to various features, which we do using 'nodes'. Nodes are nothing more than components of DTree structures. Each node carries local information, the most important items being the feature which it references, and its 'colour', which is the advice the node gives to the top node of the rule to which it belongs.

Colour is represented by the arithmetic symbol following the name of the feature in the display. As the basic connective of the system — represented by an edge between nodes — is an unless clause, the system by default alternates '+' (yes) and '−' (no) nodes. Since most features are usually not present we begin with a 'minus sign' (default values can be changed by pointing to the node and selecting 'Change Colour'). So the basic way to read a DTree rule for a feature is 'feature not present(minus sign), unless.....(+) unless (minus sign).....' to whatever depth you please.

So we can read the rule given as 'Entitled to Widow's Allowance is false unless Satisfies conditions for Widow's Allowance is true, in which case Entitled to Widow's Allowance is true unless any of (the overriding considerations) are true'.

Now we must encode a rule for 'Satisfies conditions...'. We will subsume 'woman' under 'widow' at this stage, later writing a rule to define 'widow' as a woman whose husband has died during their marriage. Examination of the Act shows that a possible encoding would see the conjunction of widowhood and husband's contribution record as being the primary condition, with other considerations being secondary: other encodings are of course possible, but this one

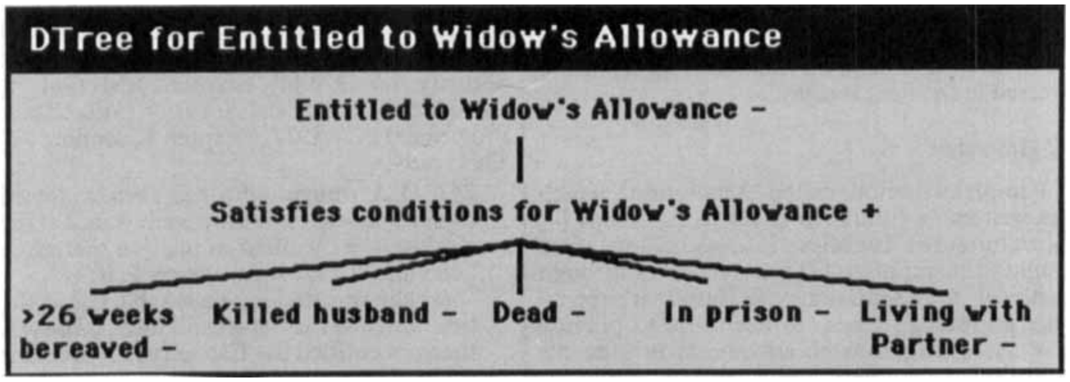
DTree for Entitled to Widow's Allowance
Entitled to Widow's Allowance −
Satisfies conditions for Widow's Allowance +
>26 weeks bereaved −
Killed husband −
Dead −
In prison −
Living with Partner −

Figure 1. *Rule for "Entitled to Widow's Allowance"*

tends to preserve the intuitive idea of putting the most probable conditions highest in the tree, and (incidentally) tends to promote efficiency of execution. Thus we would have a rule as indicated in Figure 2.

2.4 Evaluating the rule

Having built a rule, we need to be able to apply it to a case. Before this could be done seriously, explanation fragments would have to be added to the nodes. The details of this will be described later. For now, assume that it has been done, and imagine that we want to know of some particular claimant, whether they are entitled to widow's allowance. Point to a node for 'Entitled to Widow's Allowance' and select 'Run Feature'.

The inference engine will now evaluate the rule, calling other rules as needed to determine particular values, and report whether the claimant is entitled to the benefit. It will do so by searching the rule structures passing through only those nodes whose features evaluate to true; as it searches, it will emphasize, on the display, the edge along which it is searching. As each feature is first encountered, if the 'ask user?' flag of the feature is 'yes', it will ask the user for the value for the feature, it will check whether the feature owns a rule structure, and it if does, will evaluate that. As each new rule is called, it will be displayed, attached to the previous display so that it overlays the node from which it was called. If no rule exists, or if (as should not happen) the rule is actually in use further up the recursion stack, the default value will be taken. When the value of a feature has been found, it is stored: further attempts to evaluate a given feature during a given search will retrieve this stored value.

At the end of the run, Arboretum will open a window on an explanation, in which it will print a letter to the claimant, explaining what the decision was and how it was reached; although the English tends to be somewhat stilted, we consider that it gives genuine and valuable information in a form which both the user and the claimant can be expected to understand (and certainly more than claimants receive under the current manual system).

2.5 Reading rules

Let us summarise how to read a DTree rule structure. The basic units are nodes and the edges between them. An edge should always be read downwards and when connecting different colours as meaning 'unless'. Thus the most basic structure is 'hypothesis is false unless condition is true':

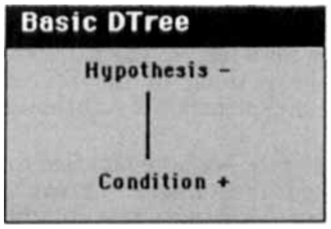

Figure 3. *Simplest possible rule*

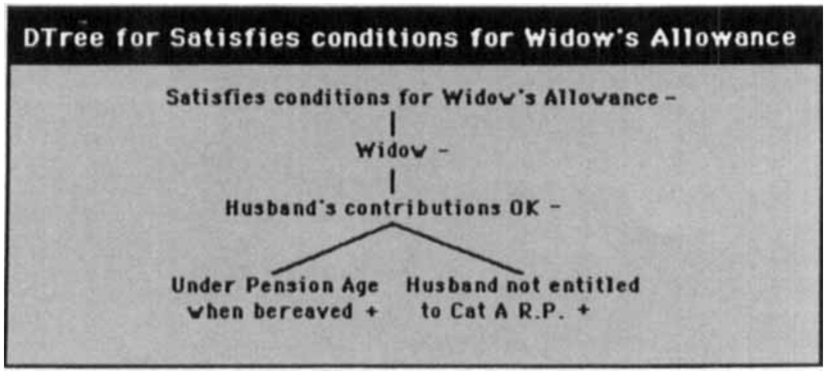
DTree for Satisfies conditions for Widow's Allowance
Satisfies conditions for Widow's Allowance -
Widow -
Husband's contributions OK -
Under Pension Age when bereaved +
Husband not entitled to Cat A R.P. +

Figure 2. *Rule for "Satisfies conditions for Widow's Allowance"*

Conjunctions are represented by columns of nodes, only the last of which has the colour to be returned if all are true and disjunctions by branches, each of which terminates in the colour to be returned if any are true. These can be combined in any fashion desired, although we consider it good practice to keep individual rule structures small. This is shown in the figure below:

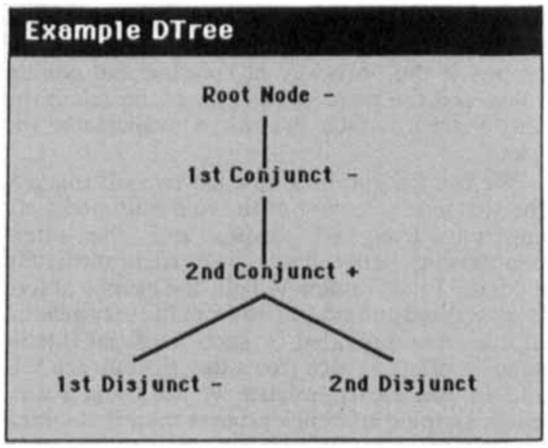

Figure 4. *Example rule, showing syntax*

The rule would read: '(rootnode) is false unless (first conjunct) is true and (second conjunct) is true, in which case it is true unless either (first disjunct) or (second disjunct) is true'.

We are not wholly satisfied by our representation of conjunction, because although logically correct, it upsets the conception of elicitation by exception: in conjunction columns, it is categorically not the case that each movement down the tree signifies a reversal of decision. From this point of view, the whole conjunction should label a single node. However, to represent it thus interferes with the full animation of the search, and this is very undesirable. We have considered several ways of representing conjunction on the display, but have yet to be fully satisfied with any of them. Perhaps the most promising is the idea of introducing a third 'colour symbol', as shown overleaf. The '&..' logically simply repeats the colour of the preceding node, but it should be clearer to the user.

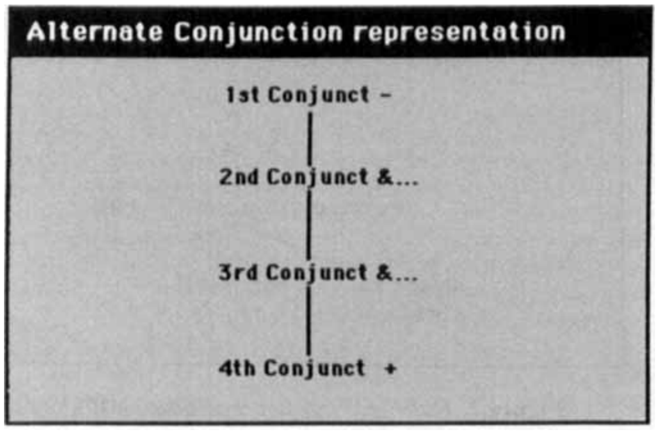

Figure 5. *Suggested alternative representation of conjunctions*

2.6 Graphical rules: discussion

In Arboretum, there is (as far as the user is concerned) no command line. All interaction with the machine — with the exception of the editing of explanation texts and typing in names of new features — is carried out with the aid of the mouse, manipulating graphs and selecting from menus. Thus the interactive manipulation of graphs is the only way of creating and editing rules, and the graphical display of the rule is the only form in which the rule is available to the user.

We feel the graphical rule has two advantages: the syntactic structure of the rule is immediately apparent from its shape, and the inter-relationships between predicates are immediately evident. Like Guidon-Watch, the graphical tool is specialised and simplified from the very general utilities now provided by such Artificial Intelligence toolkits as Kee (for a description, see [8]) and Loops. Partly because we are using a very much simpler inference process than that which underlies Guidon-Watch, we are able to get away with only one display tool.

However, this display tool actually shows the rules themselves in graphical form: Guidon's rules cannot be displayed in this way. Instead, one may look at graphs which show which rules call which others (the 'metarules window'), and at graphs which show which predicates relate to which others (the 'taxonomy window'). The inference process is animated in this latter, with current hypotheses being highlighted by boxing. Thus the display in the taxonomy window is conceptually very similar to that in our DTree Display window.

Again, the authors of Guidon appear to offer their taxonomy graph only as a single monolithic structure. This might at first appear to have the advantage that the whole graph is available at once: but in fact, unless the knowledge base is relatively trivial, it will not be possible to display the whole graph on the screen in any case.

We believe that the ability to modularise tree structures is important. Large graphs, like large chunks of code, are indigestible. Obviously, the exhaustive conditions for any given benefit, like the exhaustive taxonomy of human disease, could be displayed as a single structure: but we prefer to apply something like a structured programming methodology to knowledge-base building, breaking monolithic structures up into smaller parts. However, it is important that inter-relationships between parts should remain clear. Consequently, in designing our display, we have taken care visually to relate displays of separate trees so that the user can see immediately, from the way trees are placed, what calls what. Thus, when we open a display on the DTree of the feature behind a node, the new display is automatically positioned so that its upper left-hand corner overlays the node from which it was called.

2.7 Explanation

The explanation facility is also important to the usefulness of the system: without it, we must effectively take the decisions of the machine on faith, and in a domain with imperfect information that is dangerous; furthermore, the machine is reduced to giving yes/no answers. We can ask "is this person entitled to benefit", but not "which benefit is this person entitled to". Of course, the system with explanation can still only give yes/no decisions, because that is the nature of the logic; but now it can say "yes, this person is entitled to benefit; and the benefit they are entitled to is mobility allowance".

In addition to this facility, however, the DTree system does provide the equivalent of the more traditional 'ask how' and 'ask why' queries provided by other inference mechanisms. We can see why a question is being asked, simply by looking at the display and seeing that it is needed in the evaluation of the current rule, whose result is needed by the preceding rule, and so on back to the question originally asked. To ask how a particular value was found, after completion of a run, we can point to a node representing the feature in question and select 'How?' from the left button menu. The machine will respond by printing out a message saying whether the value was supplied by the user, evaluated from a DTree, or taken from a default. If a DTree was called, it will be displayed.

2.8 Writing the explanation

The explanation system depends on and exploits the fact that DTrees are structured through exceptions from the very general to the more abstruse and particular; and that, in consequence, any path through a rule structure follows a line of coherent argument, again from the general to the particular. Thus a sticking node on the DTree for a feature records both a decision and, by its position in the DTree, contains implicitly an explanation of why that decision was made.

Consequently, we have attached to each node in the system a text fragment to explain the consequence for the feature whose rule the node is in, if that node is a sticking node. This explanation fragment is a piece of canned text, written by the knowledge engineer. We acknowledge the criticism (made, for example, by Swartout [9]) that this approach "makes it difficult to maintain consistency between what the program does and what it claims to do" in that rules can be edited independently of the explanation fragments. But there is no reason why the system should not be modified to generate explanation fragments itself, for example by using a text macro similar to: "<feature of root-node> was found to be <colour of stick-node> because <feature of stick-node> was true". Such a macro could then be expanded immediately after a rule was edited, to provide new explanations, guaranteed to be consistent with the current form of the rule, which could then be polished by the knowledge engineer.

When writing fragments for the system as it exists, the knowledge engineer does not have to look beyond the DTree that is being edited. The task is simply to consider a node and attach text saying why the feature of the DTree obtains (or does not) when that node is conceived as the only sticking node. The structure of the system itself then ensures that in the final text, the fragment will follow sensibly from the preceding one. The only exception that we have found to this occurs when the DTrees themselves are without explanatory content. For example, consider the DTree below:

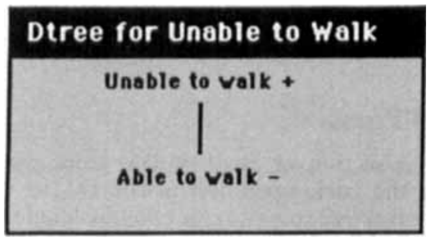

Figure 6. *Rule for "Unable to walk"*

This says that a person is unable to walk unless they are able to walk. Logically impeccable though it is, it defies sensible explanation since it is empirically vacuous. With automatic text generation, the lower node would receive the fragment "You are not unable to walk because you are able to walk" while the upper would receive "You are unable to walk because you are not able to walk". This is either a logical truth (read 'because' as 'if') or a falsehood (read 'because' causally). Clearly it does not belong in an explanation of anything. In the system as it is, the knowledge engineer is able to work around this by attaching a null explanation to each node in such trees.

2.9 Recovering the explanation

We have discussed and experimented with several algorithms for the recovery of explanations. The one we currently use works as follows: when a search results in a 'no' decision (ineligible for a benefit) then we concatenate the explanation fragments from the deepest sticking node in each successive tree on the search path. The reason is that this represents the 'nearest' that the claimant got to succeeding in the claim. This follows from the structure of the DTree, the deeper nodes represent more abstruse conditions: to reach them at all more general requirements for eligibility must have been met.

Furthermore, the information given in this explanation should be sufficient to assist in the preparation of an appeal, if the claimant feels there are further relevant facts which have not been considered — and this was, indeed, precisely our intention. It is, we think, part of the notion of relevance that it is the 'nearest miss' that should be described in such cases.

In the case of a 'yes' decision we choose the opposite approach and select the shallowest sticking node available. Partly because the claimant who succeeds is less concerned about why, but mostly because it is not relevant to describe how a long and tortuous inference path finally delivered 'yes' when a much shorter less involved one did so too. Again this seems in accord with our ordinary ideas of relevance.

It is, we think, interesting, that the structure of DTrees should so closely represent our natural ideas of relevance of explanation (at least within the present application domain).

2.10 A worked example

To provide a small worked example of an explanation generated by the system, which is yet large enough to give some flavour, let us assume a further rule to those given above:

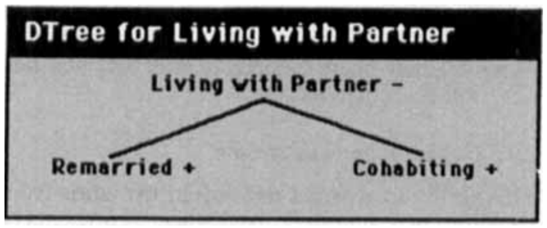

Figure 7. *Rule for "Living with Partner"*

this, indeed, is necessary to complete the encoding of the legislation fragment given. Let us further assume we have entered at least the following explanation fragments, each providing an explanation simply in the context of the rule in which it appears:

1] on the node for 'Living with Partner' in the rule for 'Entitled to Widow's Allowance', the text:

'Although you satisfy the basic conditions for eligibility for Widow's Allowance, you are not eligible, as we understand that you now have another partner.'

2] on the node for 'Remarried' in the rule for 'Living with Partner', the text:

"We understand that you have remarried."

Of course, we would also enter text fragments for all the other nodes in our rules. Now assume we are dealing with the case of a widow whose previous husband had paid his National Insurance stamps as required, and who had not reached pensionable age at the time of his death. Assume further that, within the first six months of her bereavement, our claimant has remarried. The text that would be generated would be:

> Dear [name of claimant]
>
> Although you satisfy the basic conditions for eligibility for Widow's Allowance, you are not eligible, as we understand that you now have another partner. We understand that you have remarried.
>
> Yours sincerely

There is a number of points to notice. First, we could easily have written far more friendly and less formal fragments; for example, we could have written 'we wish you all the best in your new marriage'. The formality here is simply to help in understanding the mechanical nature of the concatenation process.

Much more significantly, note that although the inference engine must have known or discovered, for example, that her previous husband had indeed been regular and conscientious about his national insurance contributions for it to have reached its conclusions, this information has been 'included out' of the explanation. It is irrelevant. It is clear that, if our claimant wished to make an appeal, her grounds for doing so would have to be that the information provided had been incorrect, and that she had not, in fact, remarried.

2.11 Explanation: a discussion

This ability to abstract relevant information from an inference engine's backtrace represents a partial solution to a problem which has been recognised by many authors in the field [10] and [11]. Early explanation systems such as Mycin [12] and Apes [13] generated explanations by listing each of the rules which had fired, with some degree of syntactic sugaring. This approach leads, as Jackson has written [10], to explanations which are 'extremely verbose and hard to follow, even in the traversal of a relatively shallow search space.'

Obviously, people have worked to improve on this, broadly by exploiting some meta-knowledge, either given as data at the knowledge engineering stage or dynamically generated at run time, which attempts to abstract out the relevant information.

As examples, Clancey [11] suggests distinguishing between 'world fact rules', which need no explanation; 'domain fact rules', which need only to be explained to non-experts; and 'causal rules', which must always be explained. Emycin [14] had a mode where its explanation was drawn from only those rules which fired successfully. Xplain [9] used an answer generator which maintained a model of what "the user can be presumed to know", which, in addition to something analogous to Clancey's distinction, exploited knowledge of what it had already told the user.

These systems are all obliged to employ devices, more or less complex, to extract a reasonable explanation from an inference trace that is itself not structured to such purposes. On the contrary the knowledge base in DTrees is, as we have argued, structured in such a way as to facilitate the generation of explanations. This is a further desirable consequence of the method of structuring knowledge by exceptions. Of course, nothing in the above is intended to denigrate the devices employed by other workers. DTree structure does not discriminate commonsense knowledge, which although critical need not be explained. Clancey's technique could be fruitfully included here. Nor is it capable, as was Swartout's system, of avoiding repetition and tautology in the explanation text.

Again, it does not address the problem of explaining to the user why the machine followed a particular strategy (see [15]): this is because, as in (most) other logic-based systems, the control strategy is fixed.

Nevertheless, the system does, at the very least, explore a presently under-used approach to the problem of explanation — namely to structure the knowledge so as to render the problem as near to trivial as possible.

3. DTrees

In this section we shall explain more systematically the basic operation of the DTree system. Our objective is to provide enough detailed information for the reader to implement a DTree system.

3.1 Philosophy

First a brief word on the philosophy of the system. The root idea is that a decision has always been made, there is always an answer available, *but one which the system is currently trying to refute.* The eventual decision is simply the *last one made*, the one that the system has failed to refute. At any point it tries to 'change its mind', and when it can no longer do so that is the decision it delivers. After all, if there is nothing as yet unexamined that could make you change your mind why deliberate further, while if there is how may you legitimately stop? This is, of course, why we aim to make the children of a node have opposite colour to the parent. The idea of an alternating 'yes/no' with decision characterised simply by its position at the end is a very old one indeed due to Thomas Hobbes [16]. The emphasis on trying to refute rather than trying to confirm is of course Popperian (*passim*, but see for example [17]).

3.2 Algorithm and data structures

A DTree system contains at any time a number of features, objects and nodes. In a Lisp implementation these are litatoms equipped with property lists. (In the Loops implementation the structures are rather different but not in any way that affects the principles of the system).

A 'feature' has the properties:
(methods DTreeRootnode default activeFlg)

where methods is the list of methods that the feature applies to see if it holds of the given object, DTreeRootnode holds a pointer to a DTree node if the feature is equipped with one (else NIL), default holds the default value of the feature, and activeFlg is simply a semaphore to prevent DTrees calling themselves.

A word more about the methods property. This may in principle hold a list of any functions that are to be applied (in the order found) with standard arguments by the feature when it is called. A feature may thus be seen as a local 'expert' whose task is to decide whether it holds of a given object by applying the list of methods it has been supplied with (this picture was suggested by Lenat [18]). However, for the sake of simplicity, we shall henceforth assume that the methods property is just the list (DoTreeDefault), and describe the algorithms accordingly.

A DTree for a feature feature is a list of nodes. A node has the properties:

(feature, colour, children, parent)

Feature points to a feature of the system, colour is either Yes or No, children point to successor nodes of the node (if any) and parent to the unique predecessor node (NIL for the origin of the DTree).

An 'object' is just a litatom. The system updates the property list of the object.

We will describe algorithms in a generic Lisp, but with concessions to readers not familiar with the language. Lisp readers should think of IF as being equivalent to COND, LOCAL to PROG, and EMPTY? to NULL. RETURN has been used occasionally unprotected by a PROG: in such instances it should be thought of as simply returning the value of its argument. The atoms TRUE and ELSE should be thought of as being bound to T, and FALSE to NIL.

We choose to present the algorithm iteratively rather than recursively for expository clarity. In order to make things clearer, we have adopted the elegant LOOP construct from Acornsoft Lisp [19]; this should be self explanatory, but briefly the list of arguments is repeatedly evaluated until the argument to an UNTIL clause evaluates to anything other than NIL, when the LOOP is exited immediately after that UNTIL clause. SET has been used rather than SETQ as more readable; and LISP: users will clearly see that many of the SETs could be avoided. We hope they will make things easier to understand.

The top-level function call is (DecideFeat feature object). This first looks on the property list of object. If it finds (feature . Yes) or (feature . No) it returns Yes or No as the case may be. Otherwise it calls the features' DTree if there is one (using DoTree) or if not, returns a default value (using Default). Default may be a complicated function (indeed it may initiate a substantial train of activity) but for present purposes it may be thought of as simply recovering 'Yes or 'No from the default property of the feature.

```
(DecideFeat (object feature)
  (IF ((GET object feature))
              {if the value is known, that's ok, do nothing}
     ((GET feature 'DTreerootnode
          (PUT object feature (DoTree object feature))
              {if a DTree exists, evaluate it — store the result on the object}
     (ELSE    (PUT object feature (Default feature]
              {Default will always succeed returning the default value of the feature. Store it on
              the object}
  (RETURN (GET object feature)))
    {now the value must be on the feature property of object — so return that}
```

DoTree is:

```
(DoTree (object feature)
  (LOCAL (result)
     (IF ((GET feature 'activeFlg) (RETURN (GET feature 'default)))
              {if the tree is already in use, don't call it again to avoid looping}
     (ELSE    (PUT feature 'activeFlg TRUE)
                  {mark the tree as in use}
          (SET 'result (DSearch object feature))
                  {search the tree}
          (PUT feature 'activeFlg FALSE)
                  {reset the marker}
          (RETURN result]
```

The function of DoTree is just clerical, the work is done by DSearch. This function induces a subtree of the DTree, the leaf nodes of which are exactly those nodes from which no exit was possible. They represent the points at which the system can no longer 'change its mind'. We call them stickNodes, and DSearch passes them to the function Decide. The algorithm for DSearch is:

```
(DSearch (object feature)
  (LOCAL (node nodesToDo newNodes children stickNodes)
        (SET 'nodesToDo (LIST (GET feature 'dTreeRootNode))
                  {origin of the DTree}
        (SET 'stickNodes NIL)
          (LOOP
            (UNTIL (EMPTY? nodesToDo))
                  {if no more to do, exit loop}
            (SET'node (POP nodesToDo))
            (SET'children (GET node 'children))
                  {get a node from nodesToDo and obtain its children...}
            (SET'newNodes NIL)
                  {..the children that can be accessed will be newNodes}
              (LOOP
                (UNTIL (EMPTY? children))
                  {if all children have been done, exit this loop}
                (SET child (POP children))
                  {get the next child}
                (IF ((EQUAL
                        (DecideFeat
                          object
                          (GET child 'feature))
                        'Yes))
                          {Run the system to discover whether the feature of the child node is
                          true of object}
                      (SET newNodes
                        (APPEND child newNodes))))
                          {..if it is then the node can be accessed and is added to newNodes,
                          and loop}
              (IF ((EMPTY? newNodes)
                  (SET 'stickNodes
                      (PUSH child stickNodes))
                          {if no new nodes were added the child is a stickNode}
                (ELSE (PUSH newNodes nodesToDo)))))
                          {otherwise the newNodes must themselves be examined for
                          stickNodes}
    (RETURN (Decide stickNodes))
    {finally, make a decision on the basis of the stickNodes found}
```

DSearch uses left to right depth first search, which corresponds well with the conceptual structure of the DTree, exploring each possibility exhaustively as it arises; but other traversals are not ruled out.

This only leaves the function 'Decide'. This simply scans the nodes submitted to it. If all of them have colour 'No' then Decide returns 'No'. But if even one has colour 'Yes' then 'Yes' is returned instead. This represents the fact that our domain rules provide sufficient conditions for access to a benefit. If you qualify by any of the criteria then the overall decision is Yes.

3.3 Logic: discussion

DTrees is a non-monotonic system (see [20]) in that what follows on the basis of limited information may no longer follow when more is given. Non-monotonicity is a feature of all systems of default reasoning, or at least all reasonable ones. For a default is only accessed in the absence of specific information and should that information be subsequently added the default will no longer be used (with consequential effects throughout the system). Predicate logic is not a non-monotonic system, so that DTrees is not a subset of predicate logic. And this raises an important question of semantics: informal understanding aside what is the meaning of a system of DTrees, *in what sense is a DTree decision a correct one?*

3.4 A semantics for DTrees

To answer this question we will do two things. First, given any rule (expressed as a bi-conditional) for deciding a sentence **Pa** we show how to construct an equivalent DTree representation. Secondly, given a DTree we shall show how to construct an equivalent sentential rule. To save clutter we will suppress reference to variables and the universal quantifier. Thus in what follows:

$$\forall x\ (Px \leftrightarrow (\neg Qx \rightarrow Rx))$$

would be represented by:

$$P \leftrightarrow (Q \vee R)$$

Suppose that the rule for a feature P is given by the bi-conditional (iff):

$$(1)\quad P <\!\!-\!\!> A(Q_1 \ldots Q_n)$$

where A represents a truth function of the features Q_i. It is well known that every such truth function is equivalent to a disjunctive normal form; that is a disjunction $D_1 \vee D_2 \vee \ldots\ldots \vee D_m$ where each Di is a conjunction of features or negated features. Suppose that D_i is in fact $Q_1 \ \& \ldots \& \ Q_k \ \& \neg\ Q_{k+1} \ \& \ \ldots \ \& \ \neg\ Q_n$. The DTree at (a) in the figure below is a DTree for the rule:

$$(2)\ P <\!\!-\!\!> Q_1 \ \& \ldots \& \ Q_k \ \& \neg Q_{k+1} \ \& \ldots \& \ \neg Q_n$$

One sees this by noting that the only way to obtain a stickNode of colour 'Yes' on this DTree is to make the right-hand side of 2) true. For you need to access Q_k and then 'stick' there.

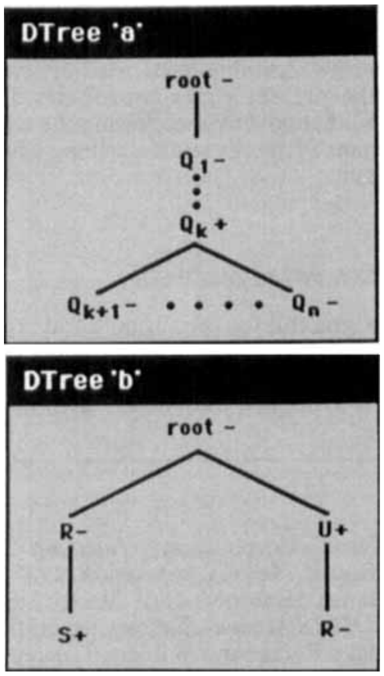

Figure 8. *Rules 'a' and 'b' used in discussion of semantics*

Now construct such DTrees for each of the disjuncts D_i and finally identify the origins of all of them. This is a DTree equivalent to the rule (1), for if (1) gives P true then the right side of (1) is true and so some disjunct as at (2) is true also. But then there is a stickNode with colour Yes. If (1) decides P false then the right side of (1) is false so none of the disjuncts as at (2) is true. But then there are no stickNodes with colour Yes on any of the subtrees. So the DTree too decides 'No'. Thus we can build a DTree to represent any rule which is given as a truth function of features.

Conversely, given a DTree one can construct a sentential representation of the rule it represents. The process is essentially the reverse of that just given. In a DTree D every node x determines a path back to the origin. Let $x_0, x_1 \ldots x_n, x$ be the path leading to x and $y_1 \ldots y_k$ be the children of x (if such there be). Then the characteristic formula for the node x is:

$$(3)\ T \ \& \ P_1 \ \& \ .. \ \& \ P_n \ \& \ P \ \& \ \neg Q_{k+1} \ \& \ .. \ \& \ \neg Q_{n+k}$$

where P_i, Q_i are the features labelling the respective nodes, and T is a sentence always true (since the origin is always accessed). We hope it is clear that x is a stickNode in the DTree just in case the characteristic formula for x is true. Now form the disjunction $V = C_1 \vee \ldots \vee C_m$ of the characteristic formulas of all nodes with colour Yes on the DTree. If D is for a feature P then the formula P **<->** V is seen to be a sentential representation of the DTree rule. As an example, the DTree in Figure b) above has the characteristic formula below:

$$(T \ \& \ R \ \& \ S) \vee (T \ \& \ U \ \& \ \neg R)$$

We are now in a position to answer precisely the question raised at the outset: 'In what sense are DTree decisions correct?' Since DTrees are equivalent to bi-conditionals, we can say that if the corresponding bi-conditional is true, and if the features assumed of the object are true, and if (finally) the default values accessed by the system are also allowed to be true, then the decision given by DTrees is true also. For it is a logical consequence of these. In short: assumptions plus used defaults plus DTree rules logically imply DTree decision. This can be shown rigorously but we shall not do so here. The important thing is to find the sentential analogues for graphical DTree structures so that we can sensibly speak of them entering into relations of logical implication at all, and this we have done.

One final point. The characteristic formula of a DTree has a rather direct Prolog representation. In fact construct clauses:

$$P <\!- P_1, \ldots P_n, \neg(Q_1), \ldots, \neg(Q_k)$$

for each of the nodes of colour Yes in the given DTree. Prolog itself guarantees that the 'if' clauses together amount to 'if and only if'. The use of negation as failure corresponds to the simple default mechanism that we have described. The standard Prolog depth first left right search happens also to be the search regime chosen for DTrees. Thus DTrees, with its structure of exceptions could be used to help construct Horn Clause representations of legislation.

4. Representing the law

The claim has been made (primarily by Marek Sergot [21] and in several unpublished notes) that logic programming is ideally suited to the sort of applications we have just described. It is of course not controversial that the language Prolog is suitable for such applications — indeed, Peter Hammond [22] has reported on the encoding of a part of the Social Security legislation in it. Sergot's claim is rather that some, or even all, legislation can be usefully represented as a collection of Horn Clauses. This is a claim not about a programming language but about the law.

We doubt that all legislation can be so represented. Some writers (see [23]) will dispute that the law can be codified as a set of rules at all. Even on a more traditional view (see [24]) which accepts the rule paradigm, we can hardly move from this to allowing that the extremely simple Horn Clause provides an adequate form for the whole of the law. Be that as it may, it is the weaker claim that a lot of law can usefully be so represented which is the one that matters here. That it can be so we do not wish to dispute (after all we saw in the last section how any DTree can be represented as a set of Horn Clauses). However, that a Horn Clause representation is a useful one seems to us more open to argument.

It is our contention that such a representation is insufficiently scrutable. Despite the enthusiasm of such as Kowalski [25] for Predicate Logic, one of the authors has been teaching it for some years and the plain fact is that as few people can manage logic as can manage mathematics (not, in fact, exactly the same few).

Furthermore, the difficulty people experience with logic occurs when they are using it in a benign environment: the problems they are set are designed to be accessible. The legislative environment is far from friendly. Our colleague Mary Whittaker has recently been translating the rules and guidelines relating to student claims for benefit into a Horn Clause like format (but Lisp based). She reports numerous difficulties: repetitions of rules in slightly different forms, discrepancies and apparent contradictions (arising generally out of case law), general lack of clarity, and the use both of identical forms of words to mean different things and different forms of words to mean the same things. She also noted that the exceptions to rules are often difficult to determine, and when they have been found it is difficult to insert all the 'nots' into the rules properly.

The idea that the law is structured through exceptions is the root of DTrees. This idea underlies the explanation mechanism we have described and makes possible the graphical interface. It solves at once the problem of floating 'nots' and, we think, generally helps the knowledge engineer structure a legislative domain.

Furthermore it is much easier to learn than is Predicate Logic. This is important. For if a piece of automated legislation is eventually to be used, then the representation of the law that it implements will have to be validated by persons with sufficient authority to do so. Such persons are likely to lack knowledge of logic and even if they don't are unlikely to be willing to become involved in a mass of Horn Clauses. But they will still need to understand properly and in depth (and if need be criticise and revise) whatever representation of the law the system uses. To appreciate the force of this point we need to recall that the knowledge engineers are *ipso facto* interpreting legislation. For they are re-writing it, and any such re-writing, however conservative in intention, is always an interpretation and requires authority.

We have argued that the DTree system, by exploiting the structure of exceptions in the law, offers definite advantages over a Horn Clause representation. But we admit that other companies favour Horn Clauses: in particular that they allow the use of many argument predicates while DTrees are (presently at least) restricted to one argument predicates. The eventual goal must be to construct a form of representation that is sufficiently powerful to represent large amounts of legislation and sufficiently scrutable to be used effectively. We hope the present paper contributes a building block to that goal by recognising the structure of exceptions in the law and describing a system that exploits it.

5. Acknowledgements

We are grateful for the support and criticism we have received from fellow members of the Alvey DHSS Large Demonstrator project, especially Heather Hopkins, Mary Whittaker and Andrew Taylor.

6. References

[1] Philip Leith, 'Legal Expert Systems: Misunderstanding the Legal Process', *Computers and the Law*, **49**, 1986.
[2] Allison Adam and Andrew Taylor, 'Modelling Analogical Reasoning for Legal Applications', in *Research and Development in Expert Systems*, iii, ed. M.A. Bramer, Cambridge University Press, 1986.
[3] Xerox Corporation: *InterLisp Reference Manual*, Xerox Corporation, 1985.
[4] Daniel Bobrow and Mark Stefik, *The LOOPS Manual*, Xerox Corporation, 1983.
[5] Mark Richer and William Clancey, 'Guidon Watch: A Graphic Interface for Viewing a Knowledge-Based System', *IEEE Computer Graphics and Applications*, 5, 11, 1985.

[6] Social Security Act 1975, HMSO, London, 1975. Social Security (Miscellaneous Provisions) Act 1977, HMSO, London, 1977.
[7] D. Neligan, *Social Security Case Law: Digest of Commissioners' Decisions*, HMSO, London, 1979.
[8] Mark Richer, 'An Evaluation of Expert System Development Tools', *Expert Systems*, **3**, 3, 1986.
[9] William Swartout, 'XPLAIN: a System for Creating and Explaining Expert Consulting Programs', *Artificial Intelligence*, **21**, 1983.
[10] Peter Jackson, 'Explaining Expert Systems Behaviour', paper given at Workshop on Explanation, Alvey IKBS Expert System Theme, 20–21 March 1986.
[11] William Clancey, 'The Epistemology of a Rule-Based Expert System — a Framework for Explanation', *Artificial Intelligence*, **20**, 1983.
[12] E.H. Shortliffe, *MYCIN: Computer Based Medical Consultations*, Elsevier, 1976.
[13] P. Hammond, 'APES: A user manual', *Research Report DOC 82/9*, Imperial College, London.
[14] W. van Melle, E. Shortliffe and G. Buchanan, 'EMYCIN: A Knowledge Engineer's tool for constructing Rule-Based Expert Systems' in *Rule Based Expert Systems*, ed. G. Buchanan and E. Shortliffe, Addison-Wesley, 1984.
[15] Diane Warner Hasling, William Clancey and Glenn Rennels, 'Strategic Explanations for a Diagnostic Consulting System', *International Journal of Man Machine Studies*, **20**, 1984.
[16] Thomas Hobbes, *Leviathan*, ed. C.B. MacPherson, Penguin, 1968.
[17] Karl Popper, *Conjectures and Refutations*, Routledge Kegan Paul, 1963.
[18] D.B. Lenat, 'AM: An Artificial Intelligence approach to Discovery as Heuristic Search', in *Knowledge Based Systems in AI*, eds. D.B. Lenat and R. Davis, McGraw-Hill, 1982.
[19] Arthur Norman and Gillian Cattell, *LISP on the BBC Microcomputer*, Acornsoft, 1983.
[20] J. McCarthy, 'Circumscription — A form of Non-Monotonic Reasoning', *Artificial Intelligence*, **13**, 1980.
[21] Marek Sergot, 'The British Nationality Act as a Logic Program', *Communications of the ACM*, **29**, 5, May 1986.
[22] P. Hammond, 'Representation of DHSS Regulations as a Logic Program', *Department of Computing Report No. 82/26*, Imperial College, London, 1983.
[23] Philip Leith, 'Clear Rules and Legal Expert Systems', paper given at the 2nd International Conference, Logica, Informatica, Dirrito, Florence, 1985.
[24] H.L.A. Hart, *The Concept of Law*, Clarendon Press, 1961.
[25] Robert Kowalski, 'Logic For Problem Solving', *Artificial Intelligence series*, **7**, Elsevier-North Holland, 1979.

About the authors

Peter Mott

Peter Mott is a lecturer in the Department of Philosophy at Lancaster University. He received his BA from Manchester University and did graduate work at the University of California, Irvine and at Warwick University. His research is concerned with the application of mathematical logic in Artificial Intelligence and especially with the development of efficient and scrutable inference techniques based on formalisms other than first order predicate logic. He has recently been seconded to the Department of Systems to work on the Alvey DHSS Large Demonstrator.

Simon Brooke

Simon Brooke is a Research Associate in the Department of Systems at Lancaster University, working on the Alvey DHSS Large Demonstrator. He received his BA from Lancaster. His research is concerned with the users' view of mechanised inference, and especially with the problem of explanation.