Law, Computers & Artificial Intelligence, Volume 2, Number 1, 1993

Argument-based Explanation of the British Nationality Act as a Logic Program

TREVOR BENCH-CAPON, FRANS COENEN & PAUL ORTON
University of Liverpool, United Kingdom

ABSTRACT Explanations are a significant component of any knowledge-based system in the legal domain. We have previously proposed a method by which explanations can be improved by making use of annotations on program clauses as to the role of the clause, and organising the explanation according to an argument schema based on that of Stephen Toulmin. In this paper we describe an application of this approach to a part of the British Nationality Act. This serves to illustrate both the practicality of making the required annotations on a legal logic program, and the gains in terms of explanation that can be achieved.

Introduction

Explanation has always been seen as an integral part of expert and knowledge-based systems. This aspect is recognised as being of particular importance when such systems are developed in the legal domain, since in that domain the way in which a conclusion was reached is of crucial importance, because legal decisions must be capable of justification, and because it is the underlying argument that must form the basis of any subsequent presentation of the case in court or before a tribunal. The traditional form of explanation provided by rule-based systems is the so-called 'how?' explanation which derives from MYCIN [1], and which is essentially a record of the goal tree produced in proving the goal requested by the user. There are, however, a number of problems with this style of explanation.

MYCIN's explanations have been the subject of much discussion. One major reason why users did not like the system as a whole was its "inadequate answers to questions". [1] One cause of this was said to be

the lack of support knowledge in MYCIN, the mechanisms by which the action of a rule follow from the premises. It is incorrect in MYCIN, and certainly in general, to assume that a user knows a rule and merely needs reminding of it. In many cases a justification for the rule, either in terms of the source from which it was derived or of some underlying model of the domain, is required. This is not provided in MYCIN, or in the subsequent KBS which have followed its approach.

Another criticism of MYCIN's explanation is the lack of context. Each question from the user is answered in the same fashion regardless of previous answers given. That is, there is no sense of a continuing dialogue between the user and the system. Jackson [2] states that "... traces of rules were extremely verbose and hard to follow, even in the traversal of a fairly shallow search space". This verbosity is worsened by the need to include in the 'how' explanation every single step of the proof, no matter how trivial.

Swartout [3] highlights the key problem when saying that providing traces of rule activations may describe program behaviour but cannot be said to justify it. The whole technique of MYCIN's explanations is system-oriented rather than user-oriented. All explanations are in terms of the representation used in the knowledge base and structured in a manner that the system finds easy to handle. Users do not, in the main, conceptualise the domain knowledge in the form of rules at all. This makes any justification of a conclusion in terms of rules unlikely to appeal intuitively to the user. Rule traces of this type are inadequate justifications because vital information as to the thinking behind the rule remains implicit in the way the knowledge base was designed and implemented.

We have argued elsewhere [4-6] that these objections can be summarised and explained in that what is returned as an explanation is a proof, whereas what the user requires is an argument, which must take account of the different role and status of the various 'premises', and must reflect these differences in the way in which the explanation is organised and presented to the user. In [6] we described a way in which this might be done, by annotating the clauses of a logic program to record the extra-logical information required to organise the explanation, and using a meta-interpreter to build up, not a simple goal tree, but a set of relations describing the computation in terms of a relational description of an argument based on the argument schema of Toulmin [7]. In this paper we will describe an application of these ideas to a specific piece of legislation: the British Nationality Act.

This Act was chosen because it had previously been implemented in what is still the best known legal logical program, described in the widely referenced paper.[8] We should emphasise that the system we describe here is not intended to be a practical system, but only an illustration of our approach to explanation. We have not represented the whole Act, nor

any of the secondary legislation associated with it. Nor have we attempted to include any of the guidelines issued to those who apply the Act, nor any interpretative material. If we were trying to build a system for actual use this material would be required, but what we have represented is enough for our present purposes. Moreover, by sticking rigidly to the Act itself, direct comparison with the system developed in Sergot et al [8] is possible.

The Meta-Interpreter

The central idea behind our view of explanation is that the clauses of logic programs play a variety of roles. Consider, for example, the rule:

old(X):- man(X), age(X,A),A>70, not(tibetan(X)).

Here it is the clause 'age(X,A)' which provides the primary *data* used in the application of the rule. 'A>70' defines a *condition* that the data must satisfy, 'man(X)' is a *sortal* that defines the class of things to which the rule applies; and 'not tibetan(X)' *qualifies* the rule by expressing an exception to the general application of the rule. We make these various roles explicit by means of annotations on the clauses of the rule. Thus:

ruleID: old(Person):-
man(Person):class, (this is the sortal)
age(Person, Age):data, (this retrieves the age)
Age > 70:cond, (this is the test on the retrieved data)
not(tibetan(Person)):qual. (this is the exception).

Only those rules that are to be explained as arguments need be annotated in this way. Others can be left as simple Prolog rules and these will show up simply as facts. This prevents the user from being presented with trivial derivations, or explanations of system-directed tasks. Similarly system-related subgoals in an annotated rule can be annotated as such, and so ignored when the explanation is presented.

Toulmin's basic argument schema, as modified in Bench-Capon et al [6] is shown in Figure 1. We use this schema to provide a framework within which we can explain the application of the rule by mapping the different roles into different components in the schema. The sortal maps onto the class; the data onto the data; the rule itself, without the sortal, but incorporating the condition, becomes the warrant; the rule with the sortal becomes the basis; the comment giving the source of or otherwise justifying the rule becomes the backing and the qualification represents the rebuttal. All this is fully described in Lowes & Bench-Capon.[9] The work described in this paper was designed to test the effectiveness of these ideas by trying the method on a substantial logic program in the legal domain. As stated above, we chose the British Nationality Act since its use in Sergot et al [8] makes it the best known example of a legal logic program.

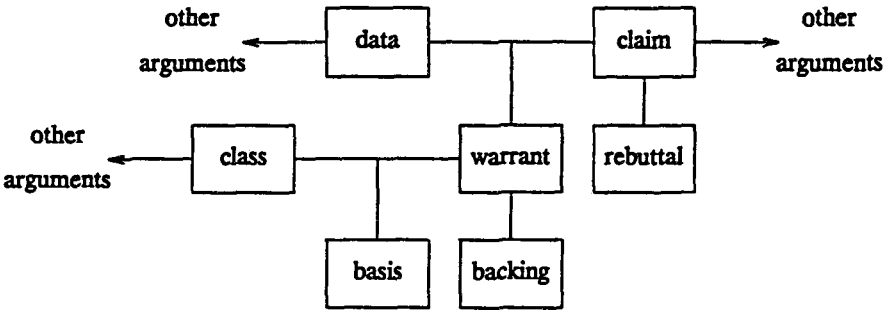

Figure 1. Toulmin's basic argument schema.

The 1981 British Nationality Act (BNA)

In 1985 a team at Imperial College London [8] constructed a logic program of much of this Act in Horn clause logic extended by negation as failure. This was embedded within the augmented Prolog system APES [11], which provides, amongst other facilities, a 'how' explanation. The Act represented an unusually self-contained piece of legislation with little case law to cloud issues. It was, thus, an ideal piece of legislation on which to try out a logic program representation, and the resulting system has been much discussed as a paradigm of the executable formalisation of legislation as a legal expert system.

The aim of our implementation was to represent the first six sections of Part 1 of the BNA in extended Horn clause logic, annotated in the manner described above, to be executed in Prolog. This provided a sufficiently large knowledge base to test and evaluate our ideas as to how such programs should explain their reasoning.

It is relevant to discuss the problems encountered in representing the Act in annotated Prolog as how the Act is represented has a bearing on the sort of explanation the system is capable of.

Representation Issues

The original idea was that the Prolog rules should be written down in the order that their corresponding sections occurred within the Act so that the rule identifiers could relate to the numbers of the sections and subsections of the Act. This was not always possible on three counts.

Firstly, not all subsections were translated into Prolog rules. For example section 1(8): "In this section and elsewhere in the Act 'settled' has the meaning given in section 50". Section 50 is a list of definitions, and 'settled' is defined by reference to the Immigration Act 1971. It was outside the scope of our intention to represent this other Act, and we

therefore made no effort to represent this definition, treating 'settled' instead as a question that would be resolved by the user. This in turn meant that no clause referring to section 50 was included in the program, since we were relying on the user understanding what is meant by 'settled'. This might be a serious defect in a practical system: but at some point the user will have to answer questions, and there is a general question of what is to be represented and what is to be left to the user. Thus in our program it is the user rather than the program that must decide if a person is settled in the UK. Clearly this precludes the system being able to give explanations or answer questions on why a person is considered to be settled in the UK. But since the user supplied the information, no explanation should be required.

Secondly, not all clauses are best represented in a single rule. This is particularly so if the clause contains a number of *OR* conditions where Horn clause logic demands that each of these is dealt with in a separate rule. In these cases the rule identifiers may not correspond exactly with the relevant section and subsection.

Thirdly, some clauses required by the program will have no counterpart in the Act. For instance there is no clause in the Act which defines what 'after commencement' means but there are four rules which define this for the purposes of the logic program. This is because the Act was written on the assumption that the people reading it had the common sense ability to reason with dates and times. 'After commencement' requires no definition for the intended reader but requires a very specific definition if it is to be interpreted by the logic program. Although we dealt with this by adding clauses to the program, since the operationalisation of this issue is clear-cut, we could have treated it as an open textured matter and referred the decision as to this predicate to the user. Such rules, essential to the program, are a source of obfuscation when they are expressed as part of the explanation, and should if possible be hidden from the user: this can be done by using the 'system' annotation.

Double Negation

A specific problem in the British Nationality Act, discussed at some length in Sergot et al [8], is the existence of some troublesome double negations. For instance section 1(2) states that a baby found abandoned in the United Kingdom will, "unless the contrary is shown be deemed ... to have been born in the UK".

Couched in more logical terms, a baby is deemed to have been born in the UK if it is *not* the case that the infant was known to have *not* been born in the UK. This double negative is, however, difficult to implement in a logic program, which interprets negation as failure. While this is satisfactory *for* the outer 'not', the inner 'not' requires classical interpretation.

Sergot et al [8] resolved this problem so as to achieve the correct results from predicates that contained a double negative by incorporating the inner 'not' in the predicate under consideration. They therefore use a predicate 'known_not_to_have_been_born_in_the_UK', and this is the form in which the question is posed to the user. This would not, however, work for us, since it would compromise the explanation.

In our program we decided that in the case of abandoned children it was better to allow for a child's country of birth to be entered explicitly as unknown and similarly for its parents to be explicitly noted as missing. This meant that a person had to be specifically identified as abandoned by the user of the program. This enabled us to draw the essential distinction between the case where a person's place of birth was known to be unknown, and the case where the place of birth simply awaited discovery. The relevant clause in our program appears:

rule1-2: uk_citizen(X, section_1-2, Date):-
found_abandoned_in(X,uk): class,
found_abandoned_on(X,Date): data,
after_commencement(Date):cond,
born_in(X,unkown):data,
parent_of(missing,X):cond.

Whilst on the subject of negation, it is worth noting that the explanation of the proof of negated statements by failure is often unsatisfactory in logic programs. Thus 'not born_in(mario,uk)' is explained by a failure to show the truth of 'born_in(mario,uk)', whereas a good explanation would rather be the *presence* of a fact such as 'born_in(mario,Italy)'. The use of rebuttals in our explanation schema provides a way in which this positive disproof can be presented, improving the explanation of this aspect.

Time in Logic Programs

The idea of a person's citizenship status changing over time is difficult to express in Horn clause logic. Contained in section 1(3) is the idea of a person becoming a British citizen or becoming settled in the UK. The notion of someone becoming a British citizen is quite clear. On the day a person's citizenship is acknowledged their status changes. However, when the question of their children's citizenship arises it is necessary to consider the source of the parent's citizenship. The fact that the parent's status has changed is as important to the Act as the status itself. It is the notion of their having gained citizenship or resident status whilst their child was resident that is crucial. That is why it is necessary to include a predicate 'becomes_a_citizen_or_settled_in(X,uk)'.

Sections 1(4) and 1(7) of the Act refer to the number of days absence from the UK a person is permitted whilst still being considered a resident; this being 90 days in one year. It might be possible to write

complex predicates concerning dates a person was absent from the UK which would settle the question of whether a person was resident or not. We, however, used a clause 'ok_on_90day_rule(X)' to keep matters simple. This again passes the decision from the program to the user who must understand what 'ok_on_90day_rule(X)' means, and is presumed to be in a position to answer the question. The same rationale also applies to the another predicate, '5_year_rule', used in the formalisation.

It is well known that there are considerable problems concerning the representation of change in logic programs and many papers have been written on this subject. This project did not attempt to tackle them, but rather avoided them by leaving them as questions requiring a decision from the user.

Outside Input into the Act

Parts of the Act, for example sections 3(1) and 6(1), call for the Secretary of State's discretion to be used. In practice, of course, this responsibility is delegated, and it is merely a device to allow the adjudicating officer to exercise discretion. The predicate thus gives the impression that outside input is required, in this case from the Home Secretary, for this rule to be applicable, but in fact this is just another question that must be resolved by the user. How this would be resolved in practice would depend on the intended use of the system: whilst it might be best for a member of the public seeking advice simply to assume that discretion would be exercised in his favour, since he can only test the question by applying, an official would probably have guidelines as to when discretion could be exercised, and in a system to support the official it would be desirable to incorporate these guidelines. We wanted our system to be neutral as to task: the result is that such predicates are incapable of further explication.

Annotating the Program

Annotations to the logic program were made whilst writing the rules themselves. This was so that the rules could be annotated whilst the rationale behind the rules and thus the rationale behind the Act was fresh in the mind of the programmer. This led to problems where the role of a clause or rule could be appreciated only in the context of other rules, so that the appropriate annotation did not become apparent until further rules had been written. Backtracking to correct these mislabelled clauses was not unusual, nor unexpected.

As examples of annotated rules, consider

```
rule1: uk_citizen(X, section_1, Date) :-
born_in(X,uk):class
born_on(X,Date):data,
```

after_commencement(Date):cond,
parent_of(Parent,X):data,
citizen_or_settled-in(Parent,uk):data.
rule1-1-5:citizen_or_settled_in(Parent,uk):-
uk_citizen(Parent, Section, Date):data.
rule-1-6:citizen_or_settled_in(Parent,uk):-
setttled_in(Parent,uk):data.

This translates Section 1(1) of the Act:

A person born in the United Kingdom after commencement shall be a British citizen if at the time of his birth his father or mother is:
(a) a British citizen; or
(b) settled in the United Kingdom.

Class Annotations

Classes were initially found difficult to label, since genuine sortals were difficult to find. Section 1 does of course refer to 'person', but that is the sortal under which all entities to which the Act applies fall. This problem with class annotations may have derived from the fact that the original argument schema was intended to deal with logic programs in general, and such programs may have concerned many different sorts of things. In the special case represented by the BNA, all rules concern persons, and so all things of interest fall under the same sortal concept. In such a case it might be thought better to ignore the class annotation altogether.

This, however, is not really the case, since the ability to distinguish two types of data can serve a useful purpose in the explanation. We therefore decided to broaden our notion of a sortal, so that the annotations could be made to work in making useful distinctions. The criterion we used for class annotation was the role that the clause would play in a future explanation. Consider the clause:

rule1-3:uk_citizen(X, section_1,D):-
entitled_to_be_registered(X):class
application_to_be_registered_made(X,D):data.

It is helpful here to separate the two conditions since the first condition is a precondition for the success of the second. Therefore to treat them as being on a par as data would be to obscure this relationship between the two conditions. We thus annotate 'entitled_to_be_registered' as a class even though it does not correspond to any reasonable sortal. Writing the program thus caused us to come to a wider view of the use of the class annotation, so that it could be used to identify this precondition-like relationship amongst clauses in the body of rules. The advantages of so doing are shown in the discussion of the example later in the paper which uses this rule. The program under consideration shows quite clearly the

need for the distinction between data and class to be drawn; it also highlights the need for a more principled approach to deciding how to make these annotations. That, however, must be a subject for further investigation.

Producing the Explanations

Method

Our idea was to present the explanation as an interactive dialogue, giving the user a sense of control and exploration. As in the conventional explanation the starting point is the claim, but instead of only being allowed to ask 'how' the claim was established, the user can ask a variety of questions so as to navigate the underlying Toulmin structure. Figure 2 shows the Toulmin structure with the components linked by arcs representing the questions that the user can ask from that point in the structure. In addition the user can terminate the explanatory dialogue at any point where he is satisfied with the explanation.

The user carries on the dialogue by the use of menu options. For example if the dialogue is presently on a warrant the user will offer the four options shown in Figure 2:

- in this case (to move to the data);
- presupposing (to move to the class);
- on account of (to move to the backing);
- thus (to end the dialogue and return to the claim).

The menu options have been chosen to provide pieces of text to connect the modules they link. The options could be in the form of questions, like 'why' or 'by what reason' commands at the claim in Figure 1. Interrogative menu options are more likely to give the user a sense of dialogue. They give the impression of a question and answer session in which the user is in control. The user may find this more satisfying than a straight traversal that, with connecting text, may read like one extremely long sentence.

To avoid swamping the user with information it may be desirable to put modules, containing several clauses, onto the screen one clause at a time. A menu option 'more' could be added to those usually present to signal to the user the presence of further clauses in that module.

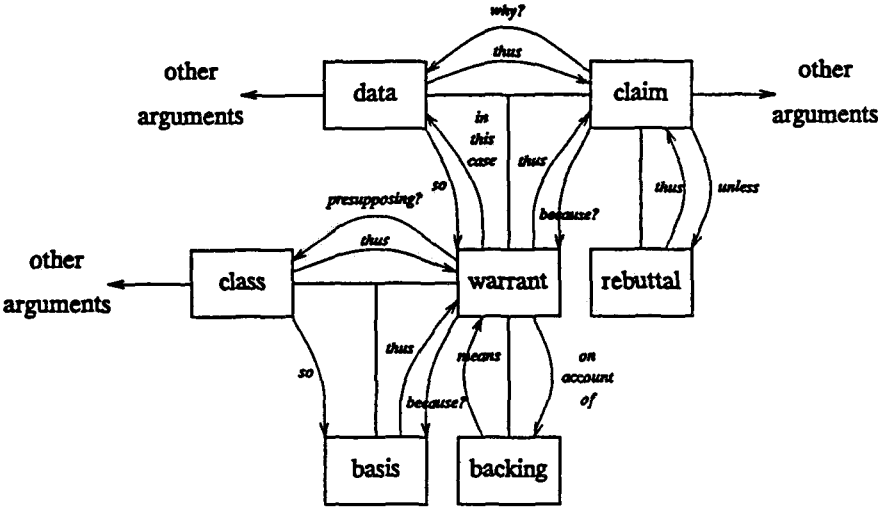

Figure 2. Menu options.

Example Explanations

We can now compare the explanations produced by this method with a conventional 'how' explanation. Suppose we run the program with the following facts:

```
application_to_be-registered_made(danny, date(23,8,1985)).
born_in(danny,uk).
born_on(danny, date(8,4,1973)).
ok_on_90day_rule(danny).
```

The relevant rules are:

```
rule1-3:  uk_citizen(X, section_1, D):-
          entitled_to_be_registered(X)                    :class,
          application_to-be_registered_made(X,D)          :data.
rule1-4:  entitled_to_be_registered(X):-
          born_in(X,Place)                                :data,
          Place = uk                                      :system,
          at_least_ten(X)                          :data,
          ok_on_90day_rule(X)                             :qual.
rule1-41: at_least_ten(X):-
          born_on(X, date(D,M,Y))                         :data,
          todays_date(date(A,B,C))                        :data,
          Y_plus_ten is Y + 10                            :system,
          after(date(A,B,C),date(D,M,Y-plus_ten))         :cond.
```

A 'How' Explanation

The 'how' explanation to Danny's citizenship will be as follows:

how(uk_citizen(danny,section_1,date(23,8,1985)))
entitled_to_be_registered(danny)
application_to_be_registered_made(danny,date(23,8,1985))

The user will probably wish to see how the first of these items were established.

how(entitled_to_be_registered(danny))
born_in(danny,uk)
at_least_ten(danny)
ok_on_90day_rule(danny)

The user may still wish to know how Danny was shown to be at least 10-years-old.

how(at least ten(danny))
born_on(danny,date(8,4,1973))
todays_date(date(8,9,1991))
1983 = 1973 + 10
after(date(8,9,1991),date(8,4,1983))

Relational Form of the Arguments

Using the argument-based explanation we first generate the argument components using the meta interpreter:

```
data(a(3), born_on(danny,date(8,4,1973)),fact).
data(a(3), todays_date(date(8,9,1991)),fact).
data(a(2), born_in(danny, uk), fact).
data(a(2), at_least_10(danny),a3).
data(a(1),application_to_be_registered_made
(danny,date(23,8,1985)), fact).

cond(a(3),after(date(8,9,1991),date(8,4,1983),fact)

warrant(a(3), [at_least_ten('X'),if,born_on('X'), date('D', 'M', 'Y')]).
 todays date(date('A', 'B', 'C')),
warrant(a(2),[entitled_to_be_registered('X'),if,born_in('X',
 uk),at_least_ten('X')]).
warrant(a(1), [uk_citizen('X', section_1,'D').
 if, application to be registered made('X', 'D')]).

presupposition(a(1),entitled_to_be_registered(danny),a(2)).

basis(a(1), [uk_citizen('X', section_1, 'D'),if,
  entitled_to_be_registered('X'),
  application_to_be_registered_made('X', 'D')]).

rebuttal(a(2), ok_on_90day_rule(danny, fact).
```

backing(a(3), rule1 – 1 – 2).
backing(a(3), rule1 – 41).
backing(a(2), rule1 – 4).
backing(a(1), rule1 – 3).

Explanation Based on the Arguments

We can generate the following dialogue (system output in italics)
uk_citizen(danny,section_1,date(23,8,1985))
Why? (this question initiates the explanation, in the same way as the 'how' question in the first example)
application_to_be_registered_made(danny,date(23,8,1985)(this offers the item annotated as data in the relevant rule)
so? (this will retrieve the whole rule – the warrant for the conclusion)
uk_citizen('X' section_1, 'D'),if,application_to_be_registered_made('X', 'D')
presupposing? (now the user fails to understand why the rule should apply to Danny, and so seeks the relevant preconditions)
entitled_to_be_reigstered(danny)
why? (this was established by the use of the previous rule, so the user seeks the data that led to this conclusion)
born_in(danny,uk) – more? (Note that there are several data items here. These are organised in order or importance, so that the user can seek more or go elsewhere it satisfied).
more (in this case the user is not satisfied)
at_least_ten(danny).
why?
born_on(danny,date(8,4,1973)) – more?
ok (the user needs no more data since he recognises that Danny is now over 10-years-old. If the user has requested more, today's date would have been given. The detailed calculation, however, would not, as the subgoals have been annotated as system because the user is considered capable of doing this unaided. The user is now taken back to the claim of that argument).
at_least_ten(danny).
so? (the user know wants to know what the consequence of accepting this argument is.)
entitled_to_be_registered('X'),if,born_in('X'uk), at_least_ten('X')
unless? (the user suspects here that there are some exceptions to this rule and so the qualification is adduced)
not (ok_on_90day_rule(danny)
ok (the user is willing to accept that Danny has not had any prolonged absences from the UK.)

Comparison of the Explanations

In the dialogue form the information is presented in more easily manageable quantities than the 'how' explanation. This enables the user to digest and appreciate the significance of each reply before deciding what further explanation is required.

The dialogue gives a much clearer impression of the role each clause plays in the arguments. From the dialogue it appears much more readily that Danny became a UK citizen on 23/8/1985 because that was the date of his application, and that Danny's entitlement to register is rather a precondition for the success of his application. This clear distinction between the contribution of the two conditions and their different effects is well shown in the dialogue but entirely obscured in the 'how' explanation.

The 'how' explanation is also packed with irrelevant information and it is mixed in liberally with the vital and significant points. Items of data such as 1983 = 1973 + 10, something that should never occur in a sophisticated explanation, and today's date, which one might expect the user to know, occur throughout. These items are examples of what it is unnecessary to present to a user. The background knowledge of the user is the crucial determinant of how much and what sort of information is needed.

Conclusion

Explanation facilities must form an important part of any knowledge-based system in the legal domain. In this paper we have described an application of a novel approach to explanations which makes use of annotation on the program clauses to organise the information presented to the user and a general schema of arguments to structure this information into a dialogue. We believe that the results have shown that the method can produce a significant improvement in the quality of explanation. The work has also led to some development of the method: during the construction of the program we came to a wider interpretation of the class annotation to distinguish different contributions made by conditions. Overall the experiment has confirmed our initial views that explanation must be more than a simple recapitulation of the computation, so that it can be put in terms oriented towards the user rather than the system, and that this requires extra logical information regarding different contributions made by the conditions in the rules.

Acknowledgements

This paper is a revised version of a paper presented at the *Third National Conference on Law, Computers and Artificial Intelligence*, held at Aberystwyth, in March 1992. We would like to thank Duncan Lowes, who wrote the original meta-interpreter, and Paul Leng for his valuable comments on an earlier draft, and Kate Williams for her comments on the paper presented at the conference.

Correspondence

Trevor Bench-Capon, Department of Computer Science, University of Liverpool, PO Box 147, Liverpool L69 3BX, United Kingdom.

References

[1] Buchanan, B. & Shortliffe, E. (1984) *Rule-based Expert Systems*, p. 355. Reading: Addison-Wesley.

[2] Jackson, P. (1986) *Introduction to Expert Systems*. Reading: Addison-Wesley.

[3] Swartout, W. (1983) XPLAIN: a system for creating and explaining expert consulting programs, *Artificial Intelligence*, 21, p. 285-325.

[4] Bench-Capon, T.J.M., Lowes, D. & McEnergy, A.M. (1990) Using arguments to explain rule-based programs, in *Proceedings of the Fifth Explanation Workshop,* University of Manchester, April.

[5] Bench-Capon, T.J.M. & Lowes, D. (1990) *Using an Argument Formalism to Improve the Explanation of Rule Based Expert Systems*. Proceedings of the Cognitiva 90, Madrid, November.

[6] Bench-Capon, T.J.M., Lowes, D. & McEnery, A.M. (1991) Argument based explanation of logic programs, *Knowledge Based Systems*, pp. 177-183.

[7] Toulmin, S. (1979) *An Introduction to Reasoning.* Basingstoke: Macmillan.

[8] Sergot, M.J., Sadri, F., Kowalski, R.A., Kriwaczek, F., Hammond, P. & Cory, H.T. (1986) The British Nationality Act as a logic program, *Communications of the ACM*, 29, pp. 370-386.

[9] Lowes, D. & Bench-Capon, T.J.M. (1990) Prolog program to model Toulmin's argumentation schema in relation to explanation of logic programs, Research Report, Department of Computer Science, University of Liverpool.

[10] British Nationality Act 1981 (1984) Chapter 61. London: HMSO.

[11] Hammond, P. & Sergot, M. (1983) A Prolog shell for logic based expert systems, in *Proceedings of Expert Systems 83,* pp. 95-104.