EXPLAINABLE (AND MAINTAINABLE) EXPERT SYSTEMS[1]

Robert Neches
William R. Swartout
Johanna Moore

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

ABSTRACT

Principled development techniques could greatly enhance the understandability of expert systems for both users and system developers. Current systems have limited explanatory capabilities and present maintenance problems because of a failure to explicitly represent the knowledge and reasoning that went into their design. This paper describes a paradigm for constructing expert systems which attempts to identify that tacit knowledge, provide means for capturing it in the knowledge bases of expert systems, and apply it towards more perspicuous machine-generated explanations and more consistent and maintainable system organization.

1. Introduction

Swartout's XPLAIN system [Swartout 83] demonstrated the feasibility of producing expert systems with enhanced capabilities for generating explanations and justifications of their behavior. XPLAIN was based on two key principles: explicitly distinguishing different forms of domain knowledge present in the knowledge base, and formal recording of the system development process. We will argue that these principles are vital both for explaining and for maintaining expert systems. This paper will propose a new paradigm for building expert systems, and consider the paradigm's implications for providing automated assistance in two tasks commonly encountered in the course of developing and using expert systems:

- Generating explanations to clarify or justify the behavior and conclusions of the system;
- Extending or modifying the system's knowledge base or capabilities;

The paradigm we are proposing, which we call the *Explainable Expert Systems* approach, calls for shifting the emphasis of knowledge engineers' efforts from procedural encoding to declarative knowledge representation. In this approach, development and use take place in an integrated support environment. Knowledge engineers and domain experts collaborate to produce a rich semantic model of the declarative and procedural knowledge of the domain. Their efforts produce a knowledge base which, augmented by advice about implementation considerations, is used to guide an automatic program writer through generation of the actual code for the expert system. The program writer maintains a record of its choice points and decisions, which constitutes the system's *development history*. The code is executed by an interpreter that maintains a record of the system's *execution history*.

All together, the knowledge base, the development history, the code, and the execution history, provide the basis for question-answering routines that allow developers and users to obtain information about the origins and rationales behind the system's code, as well as to the code itself. The availability of this extra information provides domain experts with more power to critique the system, to identify deficiencies, and to find those points in the system conception or implementation responsible for a deficiency. The availability of this extra information also provides end-users greater power to understand the abilities and limitations of the system. Thus, although the *EES* paradigm does increase initial development costs, we expect that those costs will be repaid in greater acceptance by users, easier maintenance, and a greater degree of reusability that will smooth the development of related systems later on. To support this claim, let us briefly consider what the availability of extra knowledge means for the tasks mentioned earlier: explanation and maintenance.

1.1. Explanation

As we will elaborate in section 4, the availability of information from a domain knowledge base, a development history, and an execution trace yields the opportunity to provide richer explanations of the system than are available in conventional approaches. Conventional expert systems, lacking these added knowledge sources, are restricted to explanations composed from canned text or by paraphrasing the system code. These suffer from a number of flaws. Canned text cannot anticipate, or adapt to, all possible needs. Since its maintenance is a separate **and additional** task from code maintenance, the text can quickly become invalid with respect to the true state of the system code. On the other hand, code paraphrasing is limited by the information that is represented in the code -- and even more limited by the information that is not represented. Explanation by code paraphrasing can describe actions in fairly low-level terms but it cannot, for example, describe high-level principles motivating those actions or explain why those actions instantiate some high-level principle. For example, in MYCIN, the general principle that the type of an infection may be determined using a weight-of-evidence scheme is encoded in several dozen rules, specific to particular types of infections [Szolovits 83]. The general heuristic itself is never explicitly represented, and hence is not available for explanation.

More sophisticated explanations require that design knowledge behind the system be explicitly represented, which is one of the roles served by a richer domain knowledge base and a development history. Our approach to explanation depends on a taxonomy of information goals, with *explanation strategies* associated with each information goal. An explanation strategy tells the system how to inspect the knowledge base in order to obtain information relevant to a particular goal. Question answering involves inferring information goals from questions, then applying the appropriate strategy.

[1]The research described in this paper was supported under DARPA Grant #MDA 903-81-C-0335. We would like to thank Bob Balzer, Jack Mostow, and Dave Wile for their comments on previous drafts. We would also like to thank Neil Goldman, Tom Lipkis, Norm Sondheimer, Don Voreck, and Dave Wile for helpful input on other occasions.

1.2. Maintenance

As we will elaborate in section 3.2, the use of an automatic program writer to derive code from more abstract specifications presents an opportunity to simplify the maintenance process. The need to modify a system's code generally arises for one of three reasons:

- there is an invalid assumption or principle upon which the code was based;
- an assumption or principle was valid, but the code failed to correctly instantiate it;
- additional concerns, such as ease of implementation or efficiency considerations, make an alternative method of achieving some goal preferable.

In all of these cases, the primary tasks of a maintainer are to diagnose the cause of dissatisfaction with the current system, and to locate and modify all of the relevant code. In conventional systems, since the linkage between code and higher-level principles is not explicitly represented, there can be difficulties with both tasks. When the basis for some segment of code is either invalid or inappropriately realized, it may be very hard to reconstruct from the code alone what that basis should have been. When code is rewritten or superseded, it may be very difficult to determine what other code is affected. Consider the same example mentioned above of MYCIN's weight-of-evidence scheme for determining infection types. If one wished to change this principle, dozens of rules would have to be located and modified. In the absence of any kind of pointers to those rules, it is easy to imagine some of those rules being missed, requiring multiple iterations of modification and testing to accomplish the modification.

Similar issues arise when the goal is to extend a system. Say one wanted to add knowledge about a new infection type to MYCIN. Obviously, the many rules pertaining to existing infection types could be used as examples to indicate the form of the new rules that would have to be added. Again, though, in the absence of pointers into the code, one has no easy way of making sure that all the relevant rules are located. Thus, one has no assistance in ensuring that all the necessary new rules are added, much less that they are correctly stated.

Our approach is to provide support for the diagnosis phase of maintenance through the extended explanation capabilities, and for the code modification phase through the automatic program writer. As sections 3.1 and 3.2 will show, the approach calls for the system builders to provide a knowledge base containing descriptive knowledge of how the domain works, and abstract problem-solving methods that apply to classes of problems. A *classifier* [Schmolze 83] identifies all instances of domain concepts for which a problem-solving method must be instantiated, and the automatic program writer generates code by integrating descriptive domain knowledge and problem-solving methods. Thus, for example, MYCIN's weight-of-evidence scheme for infection types would be handled by specifying methods applicable to instances of infection types, the known infection types, and (possibly) a method for integrating the results of the method instances that will be generated. The program writer would use this information to generate the appropriate specific rules for each particular infection type. Changing the principle or adding a new infection type both would be a matter of changing a small number of assertions in the knowledge base and then re-running the program writer.

2. XPLAIN: the precursor of the EES paradigm

The XPLAIN system recognized two forms of domain knowledge (factual information vs. problem solving methods) and one kind of development (refinement by a hierarchical planner). For example, when XPLAIN was used to generate a digitalis drug dosage advisor, its factual knowledge (or *"domain model"*) included assertions such as:

- High serum calcium levels can cause increased automaticity.
- Low serum potassium levels can cause increased automaticity.
- High digitalis doses can cause increased automaticity.
- Increased automaticity can cause ventricular fibrillation.
- Ventricular fibrillation is a highly dangerous condition.

This factual knowledge was augmented by problem solving methods (or *"domain principles"*), such as,

- In adjusting the drug dosage recommendation, check for factors which can cause dangerous conditions that also can be caused by administering the drug.

Applying this principle (and, of course, others) to the factual knowledge led XPLAIN to generate procedures for adjusting dosage recommendations to account for serum calcium and serum potassium levels. As it generated that implementation from the two forms of knowledge, XPLAIN recorded the steps it had taken.

Recording the derivation of the actual low-level procedures from the domain principles enabled XPLAIN-generated systems to give more principled answers to "why" questions. XPLAIN's digitalis drug dosage advisor, for example, was capable of explaining that it was asking about the patient's serum calcium level as part of adjusting the recommended dosage, *and that this was important because too high a dosage of digitalis could interact with the effects of serum calcium level to produce the dangerous condition of ventricular fibrillation.* That is, the XPLAIN-generated system could justify its request for a patient parameter both by paraphrasing the program code, and by constructing a justification for the parameter's significance based on an abstract model of the domain.

The separation of knowledge in XPLAIN also seemed to hold promise for easing the process of extending the system. Knowledge was modularized into (a) situations where patient factors could have undesirable interactions with the digitalis dosage; and, (b) problem solving knowledge governing checking for such factors and adjusting the dosage accordingly. Since XPLAIN took responsibility for applying the problem solving rules to whatever factual knowledge was given to it, programming the system to handle a new situation and generate suitable explanations for its new behaviors would require making only a few assertions to describe the added factors, rather than writing large amounts of new code that bore great similarities to existing code.

In moving beyond XPLAIN, we have identified a number of additional forms of knowledge and methods of development which are important to represent explicitly. Three forms of knowledge seem particularly important:

Trade-offs define dimensions along which the relative success or utility of a problem-solving method should be measured, along with criteria for evaluating that success or failure, and knowledge about the likely effects of each problem-solving method along those dimensions. Although the term might be taken in other ways, we mean it is a list of the strengths and weaknesses of a method with respect to general goals (rather than, say, in direct comparison to other methods).

Preferences are context-sensitive rules for ranking and selecting from alternative operators at a given choice point on the basis of their trade-offs.

Terminology refers to mappings between abstract concepts of a domain and their more concrete manifestations.

3. The EES Framework

A broad view of the EES system is shown in Figure 3-1.

The knowledge base is the foundation stone of the EES system. The *domain model* describes how the domain works. It contains, among other things, typological and causal linkages. While the domain model describes how the domain works, it does not indicate how problem solving should be done. *Domain principles* represent problem-solving strategies and are used by the program writer to drive the refinement process. *Tradeoffs* are associated with domain principles to indicate the beneficial and harmful effects of selecting a particular strategy to achieve a goal. *Preferences* are associated with goals and are used to set priorities based on tradeoffs. Mappings between abstract terms and the concepts that realize them (which had been included as part of domain principles in XPLAIN), are broken out as a separate type of knowledge to allow *terminology* to be shared across domain principles. *Integration knowledge* is used to resolve potential conflicts among knowledge sources. *Optimization knowledge* represents ways of efficiently controlling the execution of the derived expert system.

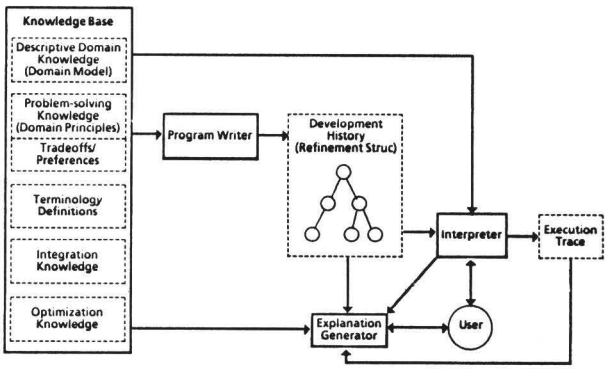

Figure 3-1: Global View of the EES Framework

The remainder of this section will illustrate the concepts introduced thus far by considering the development of a portion of a *Program Enhancement Advisor*. The Program Enhancement Advisor (PEA) is an expert system that critiques LISP programs, offering advice and assistance on stylistic improvements. The portion we will be considering seeks to improve the readability and maintainability of programs by recoding conditional expressions which use Lisp's **COND** into expressions that use alternative constructs like the **IF-THEN-ELSE** available in InterLisp's "Conversational LISP" package [Interlisp 83].

3.1. The Knowledge Base

The system's knowledge base is represented in NIKL [Moser 83]. NIKL is a refinement of KL-ONE [Brachman 78], a semantic network based representational formalism. The semantics of NIKL have been worked out sufficiently that automatic classification [Schmolze 83] is possible. Given an existing network and a new concept, the NIKL classifier automatically determines the appropriate place for the new concept in the subsumption hierarchy of the network, *based solely on the structure of that concept*.

The knowledge base contains several different kinds of knowledge, which are integrated together by the program writer to produce a working expert system. In the Program Enhancement Advisor domain, the *domain knowledge* or descriptive knowledge of how the domain works, is the knowledge of program transformations: what their applicability criteria are and what effect they will have. A simplified portion of the NIKL representation describing the transformation of a **COND** statement into a CLISP **IF-THEN-ELSE** statement is shown in Figure 3-2. Space limitations preclude explaining the NIKL representation's details, but the gist follows. The **COND** to **IF-THEN-ELSE** transformation classifies as a readability enhancing transformation because its result classifies as an easy to read construct. This latter classification holds because **IF-THEN-ELSE** is a keyword-marked construct which, in turn, is an easy-to-read construct because its parts are explicitly identified by keywords. In short, **IF-THEN-ELSE** is easy to read because each of its parts is explicitly identified by a keyword. (In section 4 we will consider how the system might mechanically create an explanation justifying the desirability of this transformation.)

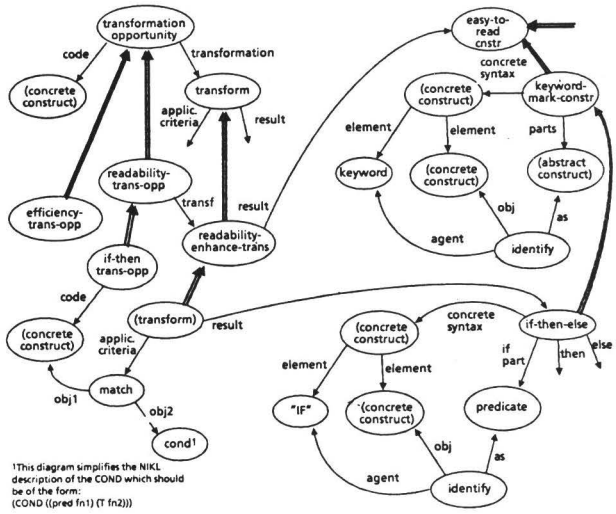

Figure 3-2: Simplified NIKL Representation of IF-THEN-ELSE Transform

PEA's *problem-solving knowledge* tells the system how to use its transformations to enhance a program. In particular, this includes strategies for scanning a program file to find places where transformations might be applied, for resolving conflicts among those possible transformation applications, and for finally applying the transformations. Plans and goals are represented in NIKL and are organized into a hierarchy by the NIKL classifier. Associated with each plan is a *capability description* which describes what the plan can do. This description is used by the system to find plans that can achieve goals.

The explicit representation of *terminological knowledge*, or the knowledge of how terms are defined and differentiated, is considerably facilitated by our use of NIKL, because it is exactly the kind of knowledge that has to be represented for the NIKL classifier to do its job. For example, in Figure 3-2, a keyword-marked construct is defined as an abstract construct whose concrete syntax has keywords that identify parts of the concrete syntax as components of the abstract construct. This structural description is used by the classifier (and the program writer as described below) to find particular instances of keyword-marked constructs.[2]

3.2. The Refinement Process

The first pass program writer creates the expert system in a top-down fashion, in this case, starting from the high level goal ENHANCE PROGRAM. As the writer implements goals, subgoals may be raised which in turn require implementation. The writer iteratively implements goals until the level of system primitives is reached. There are several means available to the system for finding implementations for goals:

Goal/Subgoal Refinement This is the familiar form of refinement by breaking a goal down into subgoals. This occurs whenever the system can find a plan that implements a goal. The system locates plans by searching up the classification hierarchy starting from the goal until it finds a plan whose capability description subsumes the goal.

Goal Reformulation into Cases When the system is unable to find a plan that implements a goal, it may reformulate that goal into several goals that can be implemented and together cover the possibilities presented by the original goal. Below, we will illustrate several uses of this kind of reformulation in the Program Enhancement Advisor. With the capability of reformulation into cases comes the need to be able to recombine the results of individual cases into an overall result for the general goal. This is where the system's integration knowledge comes into play. An example of its use in the Program Enhancement Advisor is given below.

User Directed Refinement Most current expert systems do not accept much direction from the user. Yet as expert systems move into domains where the goals are less clear cut, it becomes more important to allow the user to further specify goals. For example, in the Program Enhancement Advisor, the top-level goal of enhancing a program is underspecified. It could be that to enhance a program means to make it more readable, or it could mean to make it more efficient or maintainable. Exactly what is appropriate depends on knowledge that is outside the scope of the Program Enhancement Advisor, so it makes sense to get advice from the user to further specify such goals. However, the system must constrain the user's ability to refine goals lest he push the system beyond its capabilities. We illustrate our approach to providing the user with a constrained ability to specify goals in the example below.

From the standpoint of explanation, it is important to distinguish each of these means for implementing goals, and to record their use. ***Goal/Subgoal refinements*** indicate to the explanation facility how a low-level goal fits into the overall strategy expressed by a higher level goal. Modelling ***goal reformulation into cases*** explicitly is important because knowing that a particular goal was created due to implementation concerns usually means that that goal is unlikely to be interesting to users (but possibly quite important to system designers). Finally, the explicit modelling of user preferences afforded by ***user directed refinement*** allows the system to tailor its explanations based on known user desires.

3.3. An Example

This section outlines a portion of the steps the program writer goes through in generating the Program Enhancement Advisor. Starting from the abstract goal of enhancing a program we will show how the system moves toward generating code to scan for specific transformation opportunities. Figure 3-3 shows the development history that results from the implementation steps described below.

The system starts with the goal ENHANCE PROGRAM. The system finds a 4 step plan for performing this goal:

1. Find all applicable enhancement transformations;
2. Resolve any conflicts between candidate transformations;
3. Present recommendations to the user and ask for confirmation;
4. Act on the recommendations approved by the user.

In the development history, the writer records the implementation of the ENHANCE PROGRAM goal as a goal/subgoal refinement (see Figure 3-3).

When the system starts refining the first step of the plan, it encounters an instance of a user-directed refinement. The "dynamic-refinement" associated with the step SCAN FOR TRANSFORMATION OPPORTUNITIES specifies to the program writer that code should be created to allow the user to specify *at run-time* what kinds of enhancements should be scanned for. Since the program writer cannot predict which kinds the user will request, it will plan code to cover all kinds present in the

[2] It is worth pointing out that in the XPLAIN system, because its knowledge base did not support definition of terms and classification, terminological knowledge was represented implicitly in the domain rationale, as part of the domain principles. We now feel that this mixing of terminology with problem-solving knowledge was inappropriate. Terminology should be defined separately so that it can be consistent across domain principles.

knowledge base. Suppose it finds two kinds of enhancements, those for enhancing efficiency and those for enhancing readability. The writer would post the goals SCAN FOR EFFICIENCY-ENHANCING TRANSFORMATIONS and SCAN FOR READABILITY-ENHANCING TRANSFORMATIONS as goals to be implemented. It would also create code for interrogating the user and invoking either or both of the subgoals based on the user's desires. The system would then use its integration knowledge to combine the results of the subgoals. In this case, since the two subgoals return lists, the system uses a default strategy of appending the two lists together.

Let's consider how the writer might further refine the goal of scanning for readability enhancing transformations. There is no direct method for implementing this goal, so the program writer examines higher methods in the hierarchy. At the level of SCAN PROGRAM FOR TRANSFORMATION OPPORTUNITIES, the system finds two subconcepts that have specialized methods associated with them (see Figure 3-4). One of these scans for local transformations, that is, transformations like the COND = => IF-THEN-ELSE transform where the applicability criteria of the transform can all be verified within a single s-expression. The other scanning method scans for what we call distributed transformations, where the applicability criteria require looking at several places in the program. An example of the second type of transform would be one that verifies that it is possible to use records to replace explicit accessor functions. By examining its terminological knowledge, as expressed in NIKL, the writer determines that together these two methods cover the space of possible transforms, so the single goal of scanning for readability enhancing transforms can be re-expressed as the two goals of SCAN FOR DISTRIBUTED READABILITY-ENHANCING TRANSFORMS and SCAN FOR LOCAL READABILITY-ENHANCING TRANSFORMS. This is an instance of a goal being changed based on available domain techniques.

The system continues on in this fashion, refining general goals into increasingly more specific goals, until eventually the level of system primitives is reached. At that point the expert system is complete.

3.4. Control Issues

How would this approach be used to build expert systems with particular control structures, such as blackboard or backward-chaining architectures? We view this as a problem of specifying the interpreter that will execute code produced by the program writer. We evaluate three possible approaches below.

Our current approach is to keep the expert system interpreter very simple, and explicitly express the architecture for the system in domain principles. The program writer compiles these principles into a program simple enough for the interpreter to handle. For example, if we wanted a system to perform diagnosis using backward-chaining, we would write a principle that would say, in essence, "To determine whether a physiological state exists, conclude that it does if sufficient evidence for the physiological state exists." The causal and associational relations that would determine what was evidence for what would be expressed as domain descriptive knowledge and integration knowledge would be used to integrate the results of multiple sources of evidence.

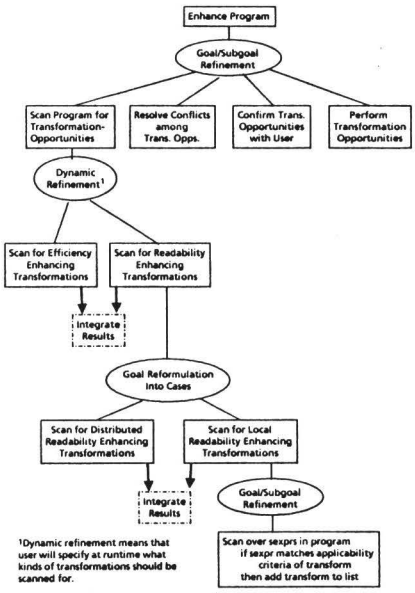

Figure 3-3: A Simplified Portion of the Development History

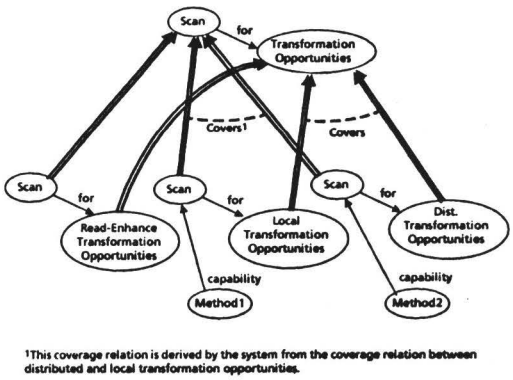

Figure 3-4: A Portion of the Method Hierarchy

This approach seems to be the most appropriate one for the two application domains we have considered, digitalis therapy and program enhancement. It allows us to cleanly and easily intermix different control strategies. Also, because the interpreter is very simple, any sophisticated features of the expert system's architecture have to be explicitly derived in the development history, so that they can be explained. The major disadvantages are that it may result in an enormous development history, and that the program writer may not be sufficiently powerful to perform all of the derivation steps.

A second alternative would be to raise the level of the interpreter so that the system primitives captured the desired architecture. The program writer would create code for this architecture. The advantages and disadvantages of this approach are just the reverse of the first approach. We have not explored this approach; we prefer the explanatory benefits of the first approach.

The most desirable (but also most difficult) alternative would have the program writer create both a high-level interpreter from

simple primitives and the code to run at that high level. A higher level interpreter would allow the development history to be smaller, and because the interpreter would be explicitly derived, its operation would be explainable. We have not yet explored this approach in detail.

4. Explanation: Procedures for Question-Answering

In previous sections we have described the types of knowledge included in the EES architecture and discussed how this knowledge is organized. In this section, we will describe the classes of questions we believe are important and discuss how the knowledge available within the EES framework enables us to provide answers to these questions.

4.1. Answering a Broader Range of Questions

In expert systems which record only the program code, and not the knowledge and reasoning required to generate that code, explanation is necessarily limited to answering questions which depend only on access to that code. Among such questions are primarily questions about *behavior*, such as:

- *How does/did the system perform <action>?*
- *How is/was parameter used?*
- *What would be the result of <parameter setting>?*

There are, however, a number of other kinds of questions that might reasonably be asked by a system builder or user:

Questions of justification, e.g.,

- *Why is the system concerned with <value, goal, or action>?*
- *Why is <goal or action> necessary (desirable, or important)?*
- *What is the significance of <result>?*

These questions all essentially seek information about the purpose underlying some aspect of the system, that is, about the relationship of that aspect to the goals of the system builder or user. Answering them involves looking at the development history to determine the domain principle(s) that generated the queried object, and from there finding further information by examining related terminology, trade-offs, and the preferences operative at the point in time under consideration.

Questions of timing or appropriateness, e.g.,

- *When did the system consider/reject <goal, action, or conclusion>?*
- *Why did it consider/reject <goal, action, or conclusion> at <time reference>?*
- *Why didn't it consider/reject <goal, action, or conclusion> at <time reference>?*

At one level, these are simply questions about the execution history of a system. Treated as such, they can be answered by techniques such as those in Davis' Teiresias system [Davis 80] that recorded triggering conditions for rules, and determined absences that prevented near-miss rules from being satisfied. However, at a higher level, the questions again deal with intentions, i.e., the reasons behind the selection conditions imposed on various knowledge items. Answering such questions may be essentially the same process as justification questions, but it may also tap knowledge that went into deriving the control aspects of the system.

Questions of definition or function, e.g.,

- *What does <term> mean?*
- *What are the effects of <action>?*
- *What is the relationship between <term, value, goal, or action> and <term, value, goal, or action>?*
- *What is the difference between <term, value, goal, or action> and <term, value, goal, or action>?*

These are questions that involve paraphrasing either the development history, the domain model, or domain principles. In each of these cases, of course, paraphrasing depends on tapping knowledge about terminology.

Questions of capabilities, e.g.,

- *What does the system know about <concept>?*
- *What factors does the system consider/ignore in concluding <conclusion>?*
- *What methods does the system use/avoid in achieving <goal>?*

Questions of this class are particularly likely to be stimulated by answers to previous questions. For example, in the case of the Xplain digitalis advisor, the answer to a justification question was that the system was interested in serum calcium levels in order to reduce the recommended dosage if the level was abnormal. This naturally leads to the question, "are there any other factors like serum calcium?" Answering such questions primarily involves searching through the domain model, examining type and causal linkages.

4.2. Information Goals and Answering Strategies

In order to devise a process model for answering the range of questions discussed above, we found it useful to categorize the questions we wished to answer into several *question types*. Associated with each question type are heuristics for determining the user's information goals. Strategies associated with a particular information goal direct the system in searching the knowledge base and producing a response which satisfies the goal. Other researchers in the area of question-answering have found it useful to identify question types and organize procedures for answering questions around these categories [Buchanan 84], [Lehnert 78], [McKeown 82]. We expect the additional step of separating question types and information goals will facilitate handling phenomena such as indirect speech acts.

A review of the strategies associated with each information goal is beyond the scope of this paper, as is a discussion of the interface which allows users to access question-answering capabilities. Here we will concentrate on a discussion of one goal and the strategy for answering it.

4.3. An Example

In this section we present a detailed examination of the strategy used for information goals of the type *Justify Result* (which are reflected in questions such as, "*Why should this advice be followed?*").

Part of the question analysis process will categorize the question as resulting from one of the information goals known to the system. Associated with the goal is the strategy to be used in generating an answer. For example, the strategy to be used in answering questions categorized as type *Justify Result* is as follows:

1. Search the development history for the <method> that produced <result>.

2. Search upward through the development history for the <goal> that this <method> is a plan for achieving (skipping those goals which are implementation concerns as described in Section 3.2)

3. State this <goal> by using its description in the domain model.

4. State how <result> is an instance of a result of achieving <goal>.

Now, we will show how this strategy is used to generate a response to an example question of type *Justify Result*. Suppose that the system had just presented the following result:

> *There are several opportunities to replace* **CONDs** *in your program with the CLISP* **IF-THEN-ELSE** *construct.*

Further suppose that the user then asks for justification of this result. Once the user's question has been recognized as being of type *Justify Result*, the system will apply the strategy described above to produce the following explanation:

> *The system scans your program looking for opportunities to enhance readability. Specifically, it is looking for constructs in your code which may be transformed into easy-to-read constructs. The system considers* **IF-THEN-ELSE** *an easy-to-read construct because it has keywords which identify its abstract components.*

In order to generate the first sentence, the system must perform the first two steps of the strategy shown above. It begins by searching through the development history looking for the <method> that produced <result>. (At this point <result> is instantiated to: COND TO IF-THEN-ELSE TRANSFORMATION-OPPORTUNITY). Referring to Figure 3-3, we see that the <method> which produced <result> is the one which scans over s-expressions to check if transformations like COND TO IF-THEN-ELSE are applicable and if so, adds to the list of enhancement opportunities.

Next, we search upward through the development history looking for the <goal> which this <method> is a plan for achieving. In performing this search, we skip over those goals which are results of the program writer's implementation concerns. Such goals frequently arise when the program writer must reformulate a single goal into several cases which together cover this goal (see Section 3.2 on goal reformulation into cases.) The development history records which goals were generated due to implementation concerns (see Figure 3-3). This enables the explainer to determine which goals must be skipped when looking for a goal that is appropriate to incorporate into an explanation to an end user. Thus, the result of the search upward in the development history is the goal SCAN FOR READABILITY ENHANCING OPPORTUNITIES.

Next, we state this goal by using its description in the domain model (step 3). As shown in Figure 3-2, a readability enhancing opportunity is a transformation whose left hand side is a construct that appears in the user's code (and which is not an easy-to-read construct) and whose right hand side is an easy-to-read construct. Thus we would generate the second clause in the explanation above.

Finally, we use the domain model in conjunction with the NIKL classifier to describe why <result>, in this case COND TO IF-THEN-ELSE TRANSFORMATION-OPPORTUNITY, qualifies as an instance of a result of having achieved the <goal> of SCAN FOR READABILITY ENHANCING OPPORTUNITIES, i.e. why <result> is an instance of a readability enhancement opportunity. First, the explainer has built into it some general world knowledge about goals. This includes the knowledge that, when trying to satisfy a goal of SEARCH FOR OBJECTS OF TYPE X, finding an object of type x is a <result> of satisfying this goal. SEARCH and SCAN are concepts defined in the domain model with SCAN being a subconcept of SEARCH. Therefore, what is left to determine is why this particular COND TO IF-THEN-ELSE TRANSFORMATION-OPPORTUNITY is an object of the desired type, i.e. why <result> is an instance of a readability enhancement opportunity. The NIKL classifier is used to provide this information. In this case, <result> is indeed a readability enhancement opportunity because its right hand side is an easy-to-read construct (and because constraints needed to satisfy ancestor concepts are also present, e.g. the left hand side is a construct which appears in the user's code and is not a good construct.)

Using the information obtained from the classifier, we can generate the last sentence in the explanation above. We wish to say why a particular **IF-THEN-ELSE** is an easy-to-read construct. To do this we look at the concept under which this construct classifies. A particular **IF-THEN-ELSE** construct will classify under the generic **IF-THEN-ELSE** definition in the domain model. Although it would not be particularly readable, we could therefore generate an explanation which states that **IF-THEN-ELSE** is an easy-to-read construct because it has a keyword IF which identifies its predicate, and do likewise for the keywords THEN and ELSE. However, since we wish to generate more abstract explanations, we have heuristics in the explainer which note such parallel structure and form generalizations. These heuristics enable the explainer to generate the last sentence in the explanation above.

5. Current Status

As of this writing, the EES framework is entering the transition from design to implementation, as is the Program Enhancement Advisor. Using programs volunteered by research programmers in our laboratory and relying on the expertise of highly skilled builders of LISP-based systems, we have identified approximately a dozen enhancements that the program ought to

be able to perform. We have set ground rules for representing concepts such as transformations, and the syntactic structure and semantic components of programming constructs. Using these rules we have completed the representation of domain and problem-solving knowledge for the **COND = =>IF-THEN-ELSE** transformation and are in the process of encoding three others. Design of a prototype version of the program writer is also almost complete. Explanation strategies have been devised for six of our 13 question types, but none are yet implemented.

6. Conclusions

In the sections above, we have argued for a new paradigm of expert system development, in which deep models separate and explicitly represent the different forms of knowledge that go into the implementation of an expert system, and in which a recorded development history is kept to trace the intertwining of those different forms of knowledge into runnable code. We have considered some particular forms of knowledge and development that seem likely to be important, discussed our design for an EES system that makes use of them, and tried to show how that system might facilitate a broader range of explanations than is possible under current expert system technology.

In addition to its implications for explanation, we believe this approach also offers other benefits related to development and maintenance. Separating the different forms of knowledge reduces the amount that has to be changed when moving to a new domain. The separation, combined with the support of the hierarchical planner, also means that less has to be done when adding new knowledge about a given domain; producing additional code to account for a new concept involves making a few assertions and re-running the program writer rather than engaging in extensive manual recoding. In addition to these maintenance-related benefits, there are also potential gains in the development process, since the discipline of explicitly specifying domain knowledge and principles is likely to make errors and inconsistencies more readily apparent.

References

[Brachman 78] Ronald Brachman.
A Structural Paradigm for Representing Knowledge.
Technical Report, Bolt, Beranek, and Newman, Inc., 1978.

[Buchanan 84] Buchanan, Bruce G. and Shortliffe, Edward H.
Rule-Based Expert Systems The MYCIN Experiments of the Stanford Heuristic Programming Project.
Addison-Wesley Publishing Company, 1984.

[Davis 80] Davis, R.
Knowledge-based systems in artificial intelligence.
McGraw-Hill, 1980.

[Interlisp 83] *Interlisp Reference Manual*
Xerox Corporation, 1983.

[Lehnert 78] Lehnert, Wendy G.
The Process of Question Answering.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1978.

[McKeown 82] McKeown, Kathleen, R.
Generating Natural Language Text in Response to Questions About Database Structure.
PhD thesis, University of Pennsylvania, 1982.

[Moser 83] M.G. Moser.
An Overview of NIKL, the New Implementation of KL-ONE.
In *Research in Natural Language Understanding*. Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1983.
BBN Technical Report 5421.

[Schmolze 83] Schmolze, J.G., and Lipkis, T.A.
Classification in the KL-ONE Knowledge Representation System.
In *Proceedings of the Eighth IJCAI*. IJCAI, 1983.

[Swartout 83] Swartout, W.
XPLAIN: A system for creating and explaining expert consulting system s.
Artificial Intelligence 21(3):285-325, September, 1983.
Also available as ISI/RS-83-4.

[Szolovits 83] Szolovits, P.
Toward more perspicuous expert system organization.
In W. Swartout (editor), *SIGART Newsletter*. ACM, 1983.
in Report on Workshop on Automated Explanation Production.