Explanations in Knowledge Systems

Design for Explainable Expert Systems

William Swartout and Cécile Paris, University of Southern California
Johanna Moore, University of Pittsburgh

AN EXPERT SYSTEM MUST KNOW HOW IT WAS DESIGNED AND PUT TOGETHER TO PROVIDE A GOOD EXPLANATION OF ITS CONCLUSIONS. IN THE EXPLAINABLE EXPERT SYSTEMS FRAMEWORK, WE CAN REPRESENT BOTH THE GENERAL PRINCIPLES FROM WHICH THE SYSTEM WAS DERIVED AND HOW THE SYSTEM WAS DERIVED FROM THOSE PRINCIPLES.

A GOOD EXPLANATION FOR AN expert system must do more than explain what the system is doing; it must also explain why the system is doing what it's doing. Such justifications of an expert system's behavior require knowledge of how that system was designed and put together. Of course, capturing the complete design rationale behind a system is a daunting task. In the Explainable Expert Systems framework (EES), we have focused on capturing those design aspects that are important for producing good explanations, including justifications of the system's actions, explications of general problem-solving strategies, and descriptions of the system's terminology. In particular, we have focused on capturing how a general principle was applied in a particular domain or a particular case. In EES, we can represent both the general principles from which the system was derived and how the system was derived from those principles.

To support good explanations, a system not only needs to capture additional knowledge, but also needs to present its explanations in a flexible and responsive manner. Conventional explanation facilities are neither flexible nor responsive because they use template-based natural-language generation techniques to produce "one-shot" explanations. Each explanation can be presented in only one way. If the user fails to understand an explanation, the system cannot offer a clarifying alternative explanation. Because such systems do not understand how their own explanations were put together, they cannot answer user queries in the context of an ongoing dialogue.

It is thus not enough simply to produce an explanation. A system must also provide elaborations or clarifications when its explanations are not understood or otherwise fail to satisfy the user.[1,2] Such an ability requires that the system understand how an explanation was produced, including what content it was intended to convey, what techniques were used to convey that content, and what alternatives were available.[2-4] Thus, in addition to knowledge of how the expert system was designed, we need another kind of design knowledge. To support dialogue, the system must understand how its explanations were designed. The EES framework (shown in Figure 1) records design knowledge both about the expert system and about explanations, thus allowing rich and responsive explanations.

From general principles to specific actions

To capture the design rationale behind an expert system, EES supports representation of general principles from which the system was designed. For example, in a medical domain, we can explicitly represent the principle that a doctor should determine if a patient is overly sensitive to a drug before administering it and adjust the dosage accordingly. This principle can then be presented to a user to explain an action derived from it.

Being able to explain this principle, however, is not enough. The system also must explain how the general principle has been applied to the problem at hand. In fact, expert-system users often already share an understanding of the general principles on which the system is based. They will, however, have questions such as: "Why is this particular finding important in diagnosing the patient?" "Why is it important to know this particular lab value?" Producing responses to these questions involves showing how the expert system's immediate concerns (such as the value of some finding) arise from applying the general principles to the particulars of the domain.

The EES framework supports exactly this sort of explanation by providing a representation for domain-independent and -dependent problem-solving principles, a representation for domain knowledge, and a way to link the general problem-solving principles and the specific domain knowledge.

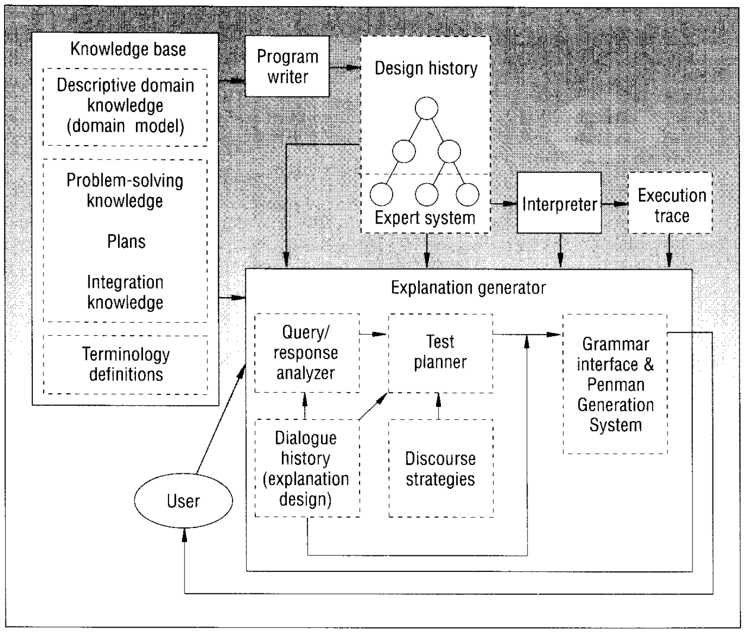

Figure 1. The Explainable Expert Systems framework.

A representation for problem-solving principles. The framework must represent general principles so that it can explain how they have been specialized to particular situations. EES uses a plan-based approach for representing problem-solving knowledge. Problem-solving principles are represented as plans, each of which has a capability description that describes what the plan is useful for (such as "diagnose faulty component" or "find the truth value of a conjunction"). During problem solving, descriptions of tasks to be performed are posted as goals. The system finds plans that might be applicable to achieving a particular goal by matching the goal against the plans' capability descriptions. Each plan also contains a method, which is a sequence of steps for achieving the goal.

A representation for domain knowledge. The system must have knowledge that describes how the domain operates. In a system for diagnosing problems in an electronic circuit, such knowledge might include the circuit schematic and descriptions of how devices that make up the circuit behave. EES uses this knowledge in applying a general principle to a specific problem. Domain models in EES are constructed using the conceptual structures of a knowledge-representation system from the KL-One family.[5] They are thus organized into a generalization hierarchy. (We have used several such representation languages during the course of the project; we are currently using Loom.[6])

In addition to domain-specific concepts, a domain model in EES also includes a number of abstract, domain-independent concepts such as Decomposable-object and Generalized-possession. These abstract concepts are the foundation on which the domain-specific portion of the model is built. (Some of these abstract concepts are predefined and come from the Penman Upper Model.[7] Differences among these concepts reflect some of the distinctions that need to be made when expressing entities from the conceptual model in natural language.)

Linking problem-solving principles and domain knowledge. EES addresses the key problem of explaining how a system's specific action follows from one of the general principles on which the system is based. The EES program writer — which derives the actual expert system from the various knowledge bases — uses two mechanisms to perform the reasoning that produces specific actions from general principles. The first mechanism is specialization. A plan's capability description might contain variables that are bound when the plan is matched against a goal. For example, in the goal "diagnose faulty component," the term "component" is a variable that can match any concept of the class Component. If this capability description is matched against the goal "diagnose faulty DECserver," then Component will be bound to DECserver. Before the method part of the plan is run, all occurrences of Component in the method are replaced by DECserver. This specialization process is recorded so that the system can explain the relation between specific actions and the more general principles they stem from.

The second mechanism is reformulation. In EES, if no plans can be found to achieve a goal, the system tries to reformulate the goal into a new, equivalent goal (or set of goals) for which plans can be found. To reformulate a goal, the system must understand it. EES represents goals as conceptual structures composed from concepts in the underlying knowledge representation. Representing goals in this way captures their semantics, allowing reformulation. Reformulation is the primary way in which domain knowledge is integrated into the process of realizing specific actions from general principles. As an example, suppose a system's domain knowledge states that Concept-A $\equiv$ Concept-B, and that no plans can be found to achieve the

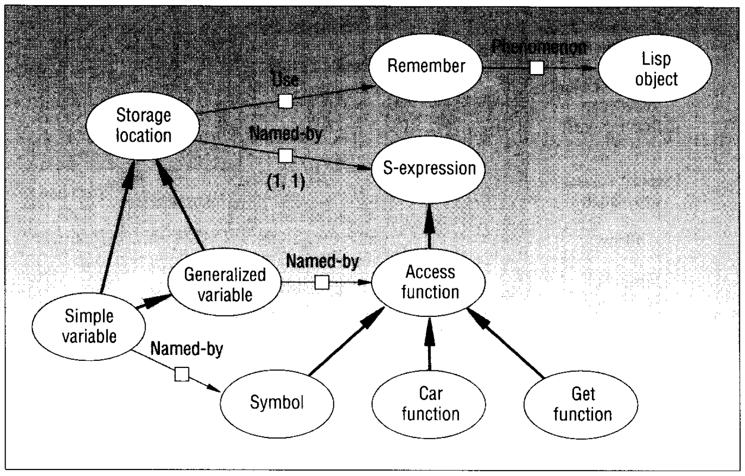

Figure 2. A portion of the domain model in the Program Enhancement Advisor.

goal "diagnose *x*," where *x* is a specialization of Concept-A. The system will use its knowledge of the equivalence to reformulate the original goal into a new one: "diagnose *y*," where *y* is a specialization of Concept-B. The resulting goal is equivalent to the original but has been reformulated so that additional candidate plans can be considered. Like specialization, the reformulation process is recorded so that the linkage can be explained.

An example

EES has been used in several prototypes, each exploring a different research issue. We use the Program Enhancement Advisor[8] here, since it is the main prototype on which the explanation work has been developed and tested. PEA is an advice system that helps users improve their Common Lisp programs by recommending transformations that enhance the user's code. PEA only recommends transformations that improve the code's "style"; it does not try to understand the program's content. After a user gives PEA a program to be enhanced, PEA asks the user what characteristics of the program should be improved. The user can choose to enhance readability, maintainability, or both. PEA then recommends transformations to enhance the program. After each recommendation, the user can ask questions about the recommendation.

If PEA had been built using a conventional approach, it would consist of a collection of transformation rules for each characteristic to be enhanced, indicating a "bad programming style" to be replaced by a "good programming style." For example, to enhance maintainability, the system would include a set of rules such as "replace Setq with Setf." The problem with this approach is that the system would not be able to justify why a particular transformation enhances the program. Instead, we want the system to explain that Setf is a more general construct than Setq, and that using it would make the program easier to maintain. Unless the rationale behind the transformations has been captured, such explanations are not possible. (Some readers — and users — might disagree with this example, feeling that replacing Setq with Setf does not improve a program's maintainability. Actually, this is a major reason for having an explanation facility in the first place: Seeing the rationale behind the recommendation is critical in such situations, because it helps the user understand why the system made its recommendation.)

In the EES framework, specific actions, such as those in the example, are derived from general principles and domain knowledge. Building PEA in the EES framework required building a domain model that describes the various concepts and relations used in the domain, as well as a plan knowledge base that includes the general principles for solving the problem of enhancing a program.

Some reasoning by EES's program writer is domain-level reasoning, such as reasoning about what program characteristics will be enhanced by a particular transformation, while other reasoning is more concerned with implementation issues, such as how the system should match particular kinds of transformations against a user's program. It is important to distinguish between these kinds of reasoning, because presenting one or the other might not always be appropriate. For example, implementation details should be skipped when presenting explanations to users, whereas they might be of great interest to system builders. In EES, the different kinds of reasoning are explicitly marked in the design record.

Domain knowledge in PEA. Domain knowledge in PEA includes definitions of the terms used in the domain, together with their relations (structural or causal). It includes descriptive knowledge about Lisp programming constructs and transformations between actual constructs. This descriptive knowledge explicitly states such facts as

(1) A program is a decomposable object owned by a user.

(2) A transformation has two parts: a left-hand side (LHS) and a right-hand side (RHS), both of which are program constructs.

(3) A maintainability-enhancing transformation is a transformation whose RHS's use is more general than its LHS's use.

(4) Storage locations are named by s-expressions (program statements in Lisp).

(5) An access function is an s-expression.

(6) A symbol is an access function.

(7) A generalized variable is named by an access function.

(8) A simple variable is named by a symbol.

(9) Setf can be used to assign a value to a generalized variable.

(10) Setq can be used to assign a value to a simple variable.

(11) The Setq-to-Setf transformation is a maintainability-enhancing transformation whose LHS is the Setq function and whose RHS is the Setf function.

(12) Local transformations and distributed transformations together cover the set of transformations. (Local transformations are those whose applicability can be determined by looking at a single s-expression, such as Setq-to-Setf, while distributed

transformations require looking at several places in a user's program before their applicability can be determined. Replacing explicit accessors, such as Car and Cadr, with record-based accessors is a distributed transformation. This distinction is important because different methods are used to search for opportunities to apply each kind of transformation.)

Using a formalism as in KL-One, PEA represents all this information as concepts organized into a generalization hierarchy, and as relations (or roles) between concepts. So, for example, Fact 1 above is represented by making the concept Transformation a specialization of the concept Decomposable-object (using the Is-a link). Fact 2 is represented by the relations LHS and RHS (both specializations of the relation Generalized-possession), both of which relate the concept Transformation to the concept Program-construct. (The representation scheme is actually more complex, allowing value and number restrictions in roles.[5,9] We have included only the elements of the representation needed for our examples.) Figure 2 shows graphically a portion of this structured inheritance network representation concerned with representing the concepts of simple and generalized variables in Lisp.

In addition to its use in deriving specific actions from general principles, this domain knowledge is used when a user asks for descriptions of system terms. Novice users often need to ask questions about terminology to understand the system's responses and to respond appropriately when answering the system's questions. Experts also might want to ask such questions to determine whether a system is using a term in the same way they do.

General principles in PEA. As mentioned before, EES represents general principles as plans that consist of a capability description (the goal the plan can achieve) and a method (a sequence of steps for achieving the goal). A capability description is written in terms of conceptual structures defined in the domain model. For example, the plan to enhance a program characteristic has the following capability description:

```
(Enhance  (Obj ((Characteristic1 Is-a
                  Characteristic)
                 Property-of Program))
           (Actor PEA-system))
```

```
(Define-plan Enhance-characteristic-of-program
   :Goal (Enhance  (Obj ((Characteristic1 Is-a Characteristic)
                         Property-of Program))
                   (Actor PEA-system))
  :Input    ((Program1 Is-a Program))
  :Output   ((Program2 Is-a Program))
  :Method   (Apply
              (Actor PEA-system)
              (Obj
                 (Set (Transformation
                        That
                          (Enhance
                             (Obj (Characteristic1
                                     Property-of Program))))))
              (Recipient Program)))
```

Figure 3. A plan to enhance a characteristic.

```
(Define-plan Apply-local-transformation
   :Goal (Apply  (Actor PEA-system))
              (Obj ((Local-transformation1 Is-a Local-transformation)
                    (LHS Is-a Program-construct)
                    (RHS Is-a Program-construct)))
              (Recipient Program))
  :Input    ((Program1 Is-a Program))
  :Output   ((Program2 Is-a Program))
  :Method   (Loop  For (S-expr1 Is-a S-expr) in Program1
                   When (Match S-expr LHS)
                   Do (Replace S-expr1 RHS)
                   Producing (Program2)))
```

Figure 4. General principle to apply local transformations.

In this example, Enhance, Program, PEA-system, and Characteristic are conceptual structures that have been defined in the domain model. Similarly, Property-of and Actor are roles defined in the domain model. (These are actually among the abstract concepts defined in the Penman Upper Model.) This plan embodies the general principle that to enhance a characteristic of a program, the system must apply transformations that enhance that characteristic. This plan is shown in Figure 3. (For clarity, we have simplified the notation for posting subgoals.)

Since the method in Figure 3 has the subgoal of applying transformations, the system needs ways to do so. Once again, those means are described by general principles. For example, PEA includes a general principle stating that to apply a local transformation to a program, the system must iterate over each s-expression in the program, trying to apply the transformation when the left-hand side of the transformation matches the s-expression under consideration. This plan is shown simplified in Figure 4.

From general plans to specific actions. The EES program writer uses a goal-refinement process to derive the actual expert system from the various knowledge bases. Given a high-level goal that represents the abstract task to be accomplished by the expert system, the program writer finds the plan that can achieve the goal and posts the appropriate subgoals. If no plans are applicable, the program writer tries to reformulate its goal. Several kinds of reformulations are possible, including

- a covering reformulation, which can be applied when a goal can be split into several subgoals that together cover the original goal;
- an individualization reformulation, which can be applied when a goal over a set of objects (such as Evaluate Symptoms) can be turned into a set of goals over individual items in the set (Evaluate Symptom-1, Evaluate Symptom-2, and so on); and
- a redescription reformulation, which can be applied when the goal can be redescribed in different terms, using terminological mappings defined in the domain model.

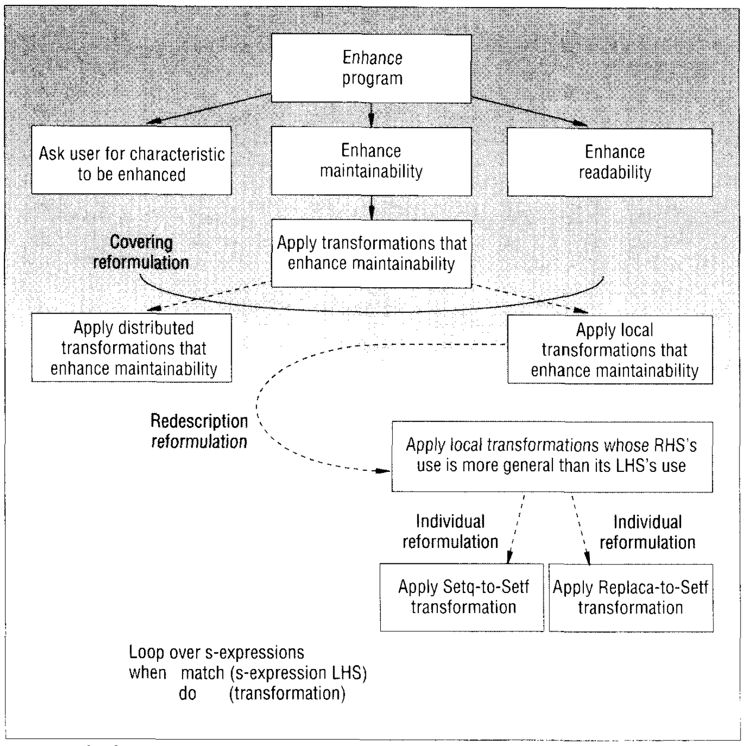
Enhance program
Ask user for characteristic to be enhanced
Enhance maintainability
Enhance readability
Covering reformulation
Apply transformations that enhance maintainability
Apply distributed transformations that enhance maintainability
Apply local transformations that enhance maintainability
Redescription reformulation
Apply local transformations whose RHS's use is more general than its LHS's use
Individual reformulation
Individual reformulation
Apply Setq-to-Setf transformation
Apply Replaca-to-Setf transformation
Loop over s-expressions
when match (s-expression LHS)
do (transformation)

Figure 5. The design history. Solid lines indicate normal goal refinement; dotted lines indicate use of reformulations.

System (1)	What characteristics of the program would you like to enhance?
User (2)	Maintainability.
...	
System (3)	You should replace (Setq x 1) with (Setf x 1).
User (4)	Why?
System (5)	I'm trying to enhance the maintainability of the program by applying transformations that enhance maintainability. Setq-to-Setf is a transformation that enhances maintainability.
User (6)	Why?
System (7)	Transformations that enhance maintainability are defined as transformations whose RHS's use is more general than its LHS's use. Setf can be used to assign a value to any Generalized-variable. Setq can only be used to assign a value to a Simple-variable.
User (8)	What is a Generalized-variable?
System (9)	Simple-variables are storage locations named by symbols. This concept can be generalized to a storage location that can be named by any accessor function. These storage locations are called Generalized-variables.
User (10)	Huh?
System (11)	For example, the "car" and "cdr" of a cons are Generalized-variables, named by the accessor functions Car and Cdr.

Figure 6. A sample dialogue.[1] "Cons" is a Lisp term denoting a cell. A cons cell is like a record with two fields. Cons cells are linked to represent lists.

Consider the goal of enhancing maintainability. The program writer first applies the plan described in Figure 3, using the fact from the domain model that Maintainability is a specialization of Characteristic. In this case, the program writer uses specialization (as opposed to reformulation) and instantiates the general plan to post the subgoal: "apply transformations that enhance maintainability." The system tries to achieve this goal and finds that no plans match it. (The plan in Figure 4 does not match the goal for two reasons: First, it can only be used to apply a single local transformation while the goal calls for applying a set of transformations. Second, the plan can apply a local transformation while the goal is to apply all transformations.)

The program writer then tries to reformulate the goal, based on the semantics of the goal itself and the domain knowledge. In this example, the system knows from the domain knowledge that local transformations and distributed transformations together cover the set of transformations (Fact 12 above). The program writer can thus perform a covering reformulation to transform its original goal into the two goals: "apply local transformations that enhance maintainability" and "apply distributed transformations that enhance maintainability." The system knows that if both of these subgoals are achieved, then the original goal will also have been achieved.

Let's now focus on the first subgoal. The system tries to find a plan to achieve this goal. It does not find a plan, but it can reformulate its goal based on its domain knowledge (Fact 3 above). This time, the program writer uses a redescription reformulation to create the subgoal: "apply local transformations whose RHS's use is more general than its LHS's use." The system still cannot find a plan matching the goal, because it has no plan capable of applying a set of transformations. Using an individualization reformulation, however, the program writer can split this goal over all instances of "local transformations whose RHS's use is more general than its LHS's use." From the domain model, the system finds such transformations — Setq-to-Setf and Replaca-to-Setf, for example. Thus, it now posts the set of subgoals: "apply Setq-to-Setf transformation," "apply Replaca-to-Setf transformation," and so on. Because the goals involve applying a single local transformation, the program writer can use the general principle shown in Figure 4 to achieve each of them by specialization. This results in posting specific subgoals, such as "replace Setq with Setf," which can be achieved with primitive actions.

The program writer records all its steps during this refinement process, including the use of reformulations. It keeps track of the type of reformulation used at each point, what aspects of the domain model were examined, and how general concepts expressed in a general plan were specialized to form a more specific plan. This results in a design history, shown in Figure 5, which indicates how specific actions were derived from general principles. This design history captures the rationale behind the specific actions and is used by the system's generation component to provide explanations of the system's behavior.

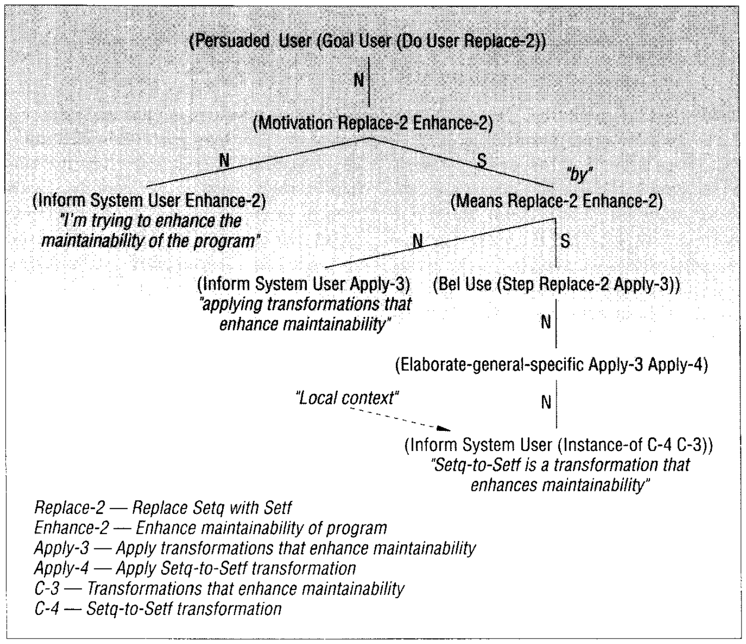
(Persuaded User (Goal User (Do User Replace-2))
N
(Motivation Replace-2 Enhance-2)
N
S
"by"
(Inform System User Enhance-2)
"I'm trying to enhance the maintainability of the program"
(Means Replace-2 Enhance-2)
N
S
(Inform System User Apply-3)
"applying transformations that enhance maintainability"
(Bel Use (Step Replace-2 Apply-3))
N
(Elaborate-general-specific Apply-3 Apply-4)
"Local context"
N
(Inform System User (Instance-of C-4 C-3))
"Setq-to-Setf is a transformation that enhances maintainability"
Replace-2 — Replace Setq with Setf
Enhance-2 — Enhance maintainability of program
Apply-3 — Apply transformations that enhance maintainability
Apply-4 — Apply Setq-to-Setf transformation
C-3 — Transformations that enhance maintainability
C-4 — Setq-to-Setf transformation

Figure 7. An explanation-text plan.

Better explanation production

Conventional expert systems use template-based techniques to generate explanations, but these suffer from a number of limitations. In particular, they cannot elaborate, clarify, or respond to follow-up questions in the context of an ongoing dialogue. Such capabilities are crucial, because experts and novices must negotiate the problem to be solved as well as a solution that the novice understands and accepts.[10,11] The EES framework offers a new approach to producing explanations — one that can participate in an ongoing dialogue and employ feedback from the user to guide subsequent explanations.

To participate in a dialogue, a system must understand the explanations it produce :. Our system explicitly plans explanations to achieve discourse goals (which denote what information should be conveyed to the user). When a discourse goal is posted, the text planner searches its library of explanation strategies, looking for strategies that can achieve the goal. The text planner selects a strategy, which might in turn post subgoals for the planner to refine. As the system plans explanations, it records its decisions in a text plan, which keeps track of any assumptions about what the user knows as well as alternative strategies that could have been used to achieve discourse goals. This text plan captures the design rationale behind the explanation by recording the discourse-goal structure of the text, the relationships between parts of the text, and an indication of the roles played by individual sentences and clauses in the overall explanation structure.[3]

After it has generated a response, the system waits for feedback from the user. The recorded text plan gives the system the dialogue context it needs to interpret the user's follow-up questions in context and to plan explanations that elaborate on previous explanations or that correct misunderstandings. Due to its central role in supporting dialogue and correcting misunderstandings, capturing the design rationale behind the explanation is as important to providing good explanatory capabilities as capturing the design rationale behind the expert system itself.

Figure 6 shows a sample dialogue with PEA. After the user gives PEA the program to be enhanced, PEA begins the dialogue by asking what characteristics of the program the user would like to improve. In this example, the user chooses maintainability. PEA then recommends transformations that would enhance the program's maintainability. The user can ask questions after each recommendation.

There are several things to note about this dialogue. First, to answer the "Why?" on Line 4, the system explains its recommendation by examining the recorded design rationale (shown in Figure 5).

Second, the "Why?" on Line 6 is ambiguous. Among other things, it could be a question about the supergoal: "Why are you trying to enhance the maintainability of the program?" In fact, this is how the Mycin system would interpret the question. However, most people interpret this "Why?" as a question about the statement that is the current focus of attention (the thing most recently said). That is, most people would interpret the question on Line 6 as: "Why is Setq-to-Setf a transformation that enhances maintainability?" Our system can make this interpretation because it understands the structure of its explanations (shown in Figure 7) and because it has heuristics for interpreting such questions based on its dialogue context.[2]

Third, to produce the answer on Line 7, the system uses the domain model and the fact that the goal was reformulated using a redescription, as recorded in the design history (Figure 7). This further illustrates how the design record supports explanations.

Finally, on Line 9, the system defines a piece of terminology. When the user indicates on Line 10 that he or she doesn't understand the definition, the system produces a follow-up explanation on Line 11 that gives examples. The system can recover from such failures because it understands and can reason about the text it produced. In this case, the system needed to know what discourse goal it was trying to achieve and how it achieved this goal (to avoid generating the same text). In addition, the system has alternative explanation strategies, selection heuristics, and recovery heuristics that let it choose the "most appropriate" strategy when an alternative explanation is required.[11] In this case, the recovery heuristic used says that when the goal that failed was to define a term, examples should be provided.

THE EES FRAMEWORK HAS BEEN implemented and the examples described here were produced by the system. We have also used the framework to construct several demonstration-size expert systems, and we are now using it to construct a larger system for diagnosing local area networks in collaboration with Digital Equipment Corporation. The system was written in Common Lisp, and the Penman text-generation system[12] was used to produce actual text from the text planner's output. The system used the NIKL knowledge-representation language,[9] but we have recently converted most of the system to use Loom,[6] a considerably more powerful representation language. We are implementing a new version of the program writer, which will use partial-evaluation techniques.

Our future research will have two distinct thrusts. First, we will investigate how to exploit EES's features for knowledge acquisition. A knowledge-acquisition facility could use EES's design record to form expectations about what new knowledge needs to be added and how that knowledge should be integrated with the rest of the system. EES's explanation facility could be used during acquisition to explain problems and to paraphrase newly acquired knowledge back to experts or knowledge engineers, letting them confirm that they added what they intended.

Our other research thrust will be to extend the range of explanations that a system can offer, so that the explanation facility will be able to provide much of a system's documentation. In addition to developing new explanation strategies, we will also need to capture additional knowledge to answer queries about how the system is used, what it is good for, or to present examples of use. Besides providing user documentation, we should also be able to use the design record to give design documentation to system builders.

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency under Grant MDA 903-81-C-0335 and under NASA Ames cooperative agreement number NCC 2-520. We thank Robert Neches and Stephen Smoliar for their contributions to the research on EES.

References

1. J.D. Moore and W.R. Swartout, "A Reactive Approach to Explanation," *Proc. 11th Int'l Joint Conf. Artificial Intelligence (IJCAI 89)*, Morgan Kaufmann, San Mateo, Calif., pp. 1,504-1,510.
2. J.D. Moore and W.R. Swartout, "A Reactive Approach to Explanation: Taking the User's Feedback into Account," in *Natural-Language Generation in Artificial Intelligence and Computational Linguistics*, C. Paris, W. Swartout, and W. Mann, eds., Kluwer Academic Publishers, Boston, 1991, pp. 3-48.
3. J.D. Moore and C.L. Paris, "Planning Text for Advisory Dialogues," *Proc. 27th Ann. Meeting Assoc. for Computational Linguistics*, Assoc. for Computational Linguistics, Morristown, N.J., 1989, pp. 203-211.
4. C.L. Paris, "Generation and Explanation: Building an Explanation Facility for the Explainable Expert Systems Framework," in *Natural-Language Generation in Artificial Intelligence and Computational Linguistics*, C. Paris, W. Swartout, and W. Mann, eds., Kluwer Academic Publishers, Boston, 1991, pp. 49-81.
5. R.J. Brachman and J.G. Schmolze, "An Overview of the KL-One Knowledge-Representation System," *Cognitive Science*, Vol. 9, No. 2, Apr.-June 1985, pp. 171-216.
6. R.M. MacGregor, "Using a Description Classifier to Enhance Deductive Inference, *Proc. Seventh IEEE Conf. Artificial Intelligence Applications*, IEEE, Piscataway, N.J., 1991, pp. 141-147.
7. J.A. Bateman et al., "The Penman Upper Model," tech. report, Univ. of Southern California Information Sciences Inst., Marina del Rey, Calif., 1990.
8. R. Neches, W.R. Swartout, and J.D. Moore, "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 11, Nov. 1985, pp. 1,337-1,351.
9. M.G. Moser, "An Overview of NIKL: The New Implementation of KL-One," in *Research in Natural-Language Understanding*, Tech. Report 5421, Bolt Beranek and Newman, Cambridge, Mass., 1983.
10. M.E. Pollack, J. Hirschberg, and B.L. Webber, "User Participation in the Reasoning Processes of Expert Systems," *Proc. Second Nat'l Conf. Artificial Intelligence (AAAI 82)*, MIT Press, Cambridge, Mass., 1982, pp. 358-361.
11. J.D. Moore, "Responding to 'Huh?': Answering Vaguely Articulated Follow-Up Questions," *Proc. Conf. Human Factors in Computing Systems*, Addison-Wesley, Reading, Mass., 1989, pp. 91-96.
12. W. Mann, "An Overview of the Penman Text-Generation System," tech. report, Univ. of Southern California Information Sciences Inst., Marina del Rey, Calif., 1983.

William Swartout started and led the Explainable Expert Systems project at the University of Southern California's Intelligent Systems Division. He is director of USC/ISI and an associate research professor of computer science at USC. His full biography and photo appear on p. 49.

Cécile Paris is the project leader of theExplainable Expert Systems project at the USC's Information Sciences Institute. Her research interests include natural-language generation, discourse, explanation, human-computer interaction, intelligent tutoring systems, machine learning, and knowledge representation. Paris received her bachelor's degree from the University of California at Berkeley and her MS and PhD in computer science from Columbia University.

Readers can reach Paris at USC/ISI 4676 Admiralty Way, Marina del Rey, CA 90292; e-mail paris@isi.edu

Johanna Moore is an assistant professor of computer science and a research scientist in the Learning Research and Development Center of the University of Pittsburgh. Her research interests include natural-language generation, discourse, explanation, human-computer interaction, planning and problem solving, intelligent tutoring systems, and knowledge representation. Moore received her BS in mathematics and computer science and her MS and PhD in computer science, all from the University of California at Los Angeles.

Readers can reach Moore at the Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260; e-mail, jmoore@cs.pitt.edu