DAN 7806

UNCLASSIFIED

DAN 7806

Technical Report

distributed by

Defense Technical Information Center
DEFENSE LOGISTICS AGENCY
Cameron Station • Alexandria, Virginia 22304-6145

UNCLASSIFIED

PROPERTY OF DACS

NOTICE

We are pleased to supply this document in response to your request.

The acquisition of technical reports, notes, memorandums, etc., is an active, ongoing program at the Defense Technical Information Center (DTIC) that depends, in part, on the efforts and interests of users and contributors.

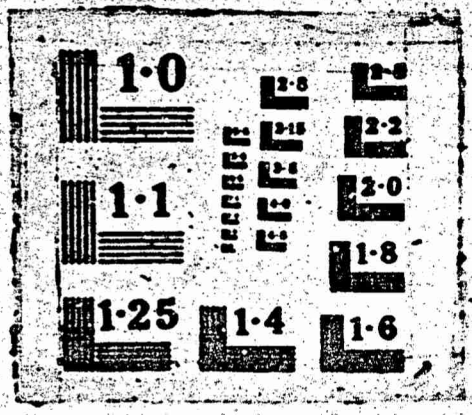
Therefore, if you know of the existence of any significant reports, etc., that are not in the DTIC collection, we would appreciate receiving copies or information related to their sources and availability.

The appropriate regulations are Department of Defense Directive 3200.12, DoD Scientific and Technical Information Program; Department of Defense Directive 5200.20, Distribution Statements on Technical Documents (amended by Secretary of Defense Memorandum, 18 Oct 1983, subject: Control of Unclassified Technology with Military Application); Military Standard (MIL-STD) 847-B, Format Requirements for Scientific and Technical Reports Prepared by or for the Department of Defense; Department of Defense 5200.1R, Information Security Program Regulation.

Our Acquisition Section, DTIC-DDAB, will assist in resolving any questions you may have. Telephone numbers of that office are: (202)274-6847, 274-6874 or Autovon 284-6847, 284-6874.

FEBRUARY 1984

☆ U.S. Government Printing Office: 1985—461-169/36024


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·2

AD-A174 505

An Explanation Facility for the ROSIE® Knowledge Engineering Language

Donald A. Waterman, Jody Paul,
Bruce Florman, James R. Kipps

DTIC
ELECTE
DEC 2 1986
S D
A

DTIC FILE COPY

Rand

86 12 02 106

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030; and by The RAND Corporation as part of its program of public service.

Library of Congress Cataloging in Publication Data

An explanation facility for the ROSIE knowledge engineering language.

"R-3406-ARPA."
Bibliography: p.
1. ROSIE (Computer system) I. Waterman, D. A. (Donald Arthur) II. United States. Defense Advanced Research Projects Agency. III. Title.
Q336.E96 1986 006.3'3 86-21947
ISBN 0-8330-0759-9

The RAND Publication Series: The Report is the principal publication documenting and transmitting RAND's major research findings and final research results. The RAND Note reports other outputs of sponsored research for general distribution. Publications of The RAND Corporation do not necessarily reflect the opinions or policies of the sponsors of RAND research.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

R-3406-ARPA/RC

An Explanation Facility for the ROSIE® Knowledge Engineering Language

Donald A. Waterman, Jody Paul,
Bruce Florman, James R. Kipps

September 1986

Prepared for the
Defense Advanced Research
Projects Agency

DTIC
ELECTE
S DEC 2 1986 D
A

Rand
1700 MAIN STREET
P.O. BOX 2138
SANTA MONICA, CA 90406-2138

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREFACE

Under the sponsorship of the Information Processing Techniques Office of the Defense Advanced Research Projects Agency (DARPA), Rand has been studying the problem of developing explanation mechanisms for expert systems. This research has focused on an explanation facility for ROSIE,[1] a knowledge engineering language developed by Rand and supported by DARPA funding.

This report describes the design and implementation of XPL (eXPlanation faciLity), a facility designed to work with ROSIE and integrated into the ROSIE 2.4 programming environment. XPL is implemented in Interlisp and operates on Xerox 1100 series workstations. The techniques used in this explanation facility are an extension of the direct paraphrasing methods used by many rule-based expert systems, i.e., they explain how a conclusion is reached by displaying the computational history on which it is based.

The audience for this report includes knowledge engineers, artificial intelligence researchers and others involved in the design and development of expert systems, and those desiring an intuitive understanding of XPL mechanisms.

This research has been conducted as part of the Information Processing Systems program of Rand's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Additional support for this study was provided by The Rand Corporation from its own funds.

[1]ROSIE is a trademark and service mark of The Rand Corporation.

QUALITY INSPECTED 4

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A1

SUMMARY

This report describes the design and implementation of an explanation facility for the ROSIE knowledge engineering language, an expert system building environment developed at The Rand Corporation. This explanation facility, called XPL (eXPlanation facility), has been implemented and integrated into the ROSIE 2.4 programming environment, creating an enhanced version of ROSIE that facilitates the development of an explanation capability during expert system building.

An explanation facility is critical in the construction and utilization of an expert system, since it provides the mechanisms by which the expert system can explain its performance. These mechanisms typically show the user how the expert system reached a given conclusion and why the system believes the conclusion is justified.

With XPL, the system builder can design expert systems capable of explaining their reasoning processes both during and after a computation. This explanation is based on a modified inference chain display. The approach consists of marking those rules and rulesets which may be of interest to the user and then generating explanations based on only those rules. This technique helps control the size and complexity of inference information stored as a history of the computation.

As the rules and rulesets are marked, they may also be assigned specific text patterns or executable procedures. During the explanation process, these patterns and procedures are combined with the current context to enhance the clarity of explanations, and they are used in conjunction with the inference chain display to explain how the system reached particular conclusions.

XPL provides the user with answers to the following kinds of questions:

1. Why was this conclusion reached?
2. Why did the system ask for this particular information?
3. Where did this rule and these data come from, i.e., how did they get added to the knowledge base?
4. What is the purpose of this rule or ruleset?
5. How can you justify the use of this rule?

XPL answers the first three questions by describing the inference chain leading to a conclusion, that is, the reasoning steps used. It also describes the rulesets and rules comprising that chain, and the

individual premises comprising the rules. XPL answers the last two questions by using patterns (text annotations containing variables) and procedures (executable rulesets) that are associated with the rules and rulesets. For problem domains that have a well-understood deep model of the mechanisms underlying the surface rules, the procedures can be used to generate sophisticated explanations or justifications describing why a particular rule is valid and why its use is appropriate in the current context. For problem domains that have poorly formalized or no deep models, the text patterns associated with the rules can be used to produce the needed explanations.

The user may select the level of detail to be provided by exercising different options when requesting an explanation. For example, the user may request a general explanation in terms of rulesets and their objectives, a more specific explanation in terms of rules and their premises in the current context, or specific examples based on patterns or procedures that make use of current context.

XPL also provides special mechanisms, such as window-creating commands and a menu package, that help the expert system builder tailor the explanation facility to his or her needs. With the window-creating commands, which can be written directly in ROSIE, the system builder can easily define, shape, and feed explanation windows. With the menu package, the system builder can easily create input/output (I/O) routines to operate within the windowing scheme.

XPL could be improved by the inclusion of graphics, monitoring, and editing features. The facility needs a sophisticated graphics capability for displaying the current state of the computation in the form of an inference tree, with a zoom capability for varying the level of detail in the display. Automatic "explanation monitoring"—a menu of specialized windows that continually answer particular questions—would help the system builder define window, question, and monitoring capabilities. And a special refinement mechanism would help users change facts or edit rules displayed during an explanation. They could then easily back up and rerun the expert system to determine the effects of minor changes on system operation. Such a mechanism would be extremely useful for refining fledgling knowledge bases and for sensitivity analyses of particular applications.

CONTENTS

I. INTRODUCTION

This report describes the design and development of an explanation facility for ROSIE (Rule-Oriented System for Implementing Expertise), a knowledge engineering language developed at The Rand Corporation (Sowizral and Kipps, 1985). This facility, called XPL (eXPlanation faciLity), has been implemented and integrated into the ROSIE 2.4 programming environment, creating an enhanced version of ROSIE that allows an explanation capability to be developed during the building of the expert system. With XPL, the system builder can design expert systems capable of explaining their reasoning processes both during and after a computation. We emphasize that XPL is the result of an initial development attempt; it is not a polished explanation package, and it could benefit from additional features, as discussed in Sec. IV.

BACKGROUND

Rand has had a long and productive history of work in artificial intelligence (AI) and expert systems (Klahr and Waterman, 1986), including both basic research and the application of that research to business and military problems. The current ROSIE system is the culmination of more than a decade of work in the development of knowledge engineering languages.

The RITA (Rand Intelligent Terminal Agent) system (Anderson et al., 1977; Anderson and Gillogy, 1976) was a successful first attempt at making programming languages easier to use and understand. In RITA, an English-like syntax was applied to rule-based programming, resulting in a language that was relatively easy to understand and use. RITA demonstrated that stylized English could be used to express procedural knowledge in a rule-based language. With only a modest introduction to RITA's syntax, even novice users could critique a ruleset productively.

RITA served as a foundation for the early work on ROSIE (Fain et al., 1981, 1982). ROSIE's elaborate English-like syntax and integration of procedural and rule-based knowledge constituted a significant improvement over RITA. A number of interesting expert systems have been built with ROSIE, including TATR (Callero et al., 1982, 1984), a system for tactical air targeting, and Adept (Taylor, 1985) a system for battle management.

THE EXPLANATION PROBLEM

The acceptability of expert systems in the user community depends not only on the quality of the systems' performance but also on how easily they can be built and used. One of the critical factors in the construction and utilization of an expert system is the mechanism it uses to explain its performance. This mechanism, called an explanation facility, shows the user how the expert system reached a given conclusion and why it believes the conclusion is justified. This ability to use and manipulate self-knowledge is a unique aspect of expert systems. By applying self-knowledge, the expert system can analyze its own operation and thus understand how and why it reaches particular conclusions.

One of the first expert systems to use an extensive explanation mechanism was MYCIN (Shortliffe, 1976), a medical diagnosis system that used a history of rule firings as the basis for explanation. Both individual rules and sequences of rules were used to explain how the system reached its conclusions. Direct paraphrasing of this type is the most widely used approach to explanation in contemporary expert systems. More complex explanation methods that are under development also use knowledge that is not required by the performance program for completion of the task. This additional knowledge may be meta-knowledge that describes high-level strategies used by the system or deep knowledge that represents fundamental principles or causal relationships among objects in the system (Clancey, 1984). This type of explanation has been tested in medical and other applications that have well-understood underlying models (e.g., electronics and engineering).

Although the explanation facilities in most current expert systems are somewhat primitive, they have proved to be crucial to system operation. They speed system development by assisting in debugging, testing, and refinement of the expert system; and they make the system acceptable to users by inspiring confidence in its performance and reasoning processes. Explanation benefits all types of users, from the system designer who needs help with debugging to the end user who is trying to decide whether or not to take the advice of the expert system.

Developing an explanation facility for ROSIE was more difficult than developing one for a typical rule-based language, such as EMYCIN (van Melle et al., 1984) or M.1 (D'Ambrosio, 1985), principally because ROSIE is both rule-based and procedure-oriented, allowing complex nested procedures that can be written with a variety of sophisticated syntactic constructs. Thus the technique of storing a history of all computations (and later trying to unravel the result) was

not appropriate. This technique works well with knowledge engineering languages that are simply sets of *If-Then* statements, but not with more complex languages like ROSIE.

Although ROSIE's inherent complexity makes the development of explanation mechanisms more difficult, its English-like syntax actually facilitates explanation. In ROSIE, a system builder can write code so readable that the system can use it directly to help explain the meaning and content of a rule even to non-programmers.

DESIGN PHILOSOPHY AND OVERVIEW

The type of explanation mechanism used by most expert systems is the *inference chain display*, in which the expert system displays a chain of rules leading to a particular conclusion (Waterman, 1986). This approach provides a very specific explanation with considerable detail. However, when used in isolation, inference chain displays may prove unsatisfactory, because the amount of detail is likely to be overwhelming, and the rule itself may not be an adequate justification for its use.

In XPL, we use a modified inference chain display, which we call *selectable paraphrasing*. Here, the rules and rulesets that may be of interest to the user are marked, and explanations are generated based on only those premarked rules. This technique helps control the size and complexity of inference information stored as a history of the computation. As the rules and rulesets are marked, they may also be assigned specific text patterns or executable procedures. During the explanation process, these patterns and procedures are instantiated by the current context and are used in conjunction with the inference chain display to explain how the system reached particular conclusions (see Sec. III).

The user may select the level of detail XPL provides by exercising different options when requesting an explanation. For example, the user may request a general explanation in terms of rulesets and their objectives, a more specific explanation in terms of rules and their instantiated premises, or specific examples based on instantiated patterns or procedures. XPL will provide answers to the following kinds of questions:

- Why was this conclusion obtained?
- Why did the system ask for this particular information?
- Where did this rule and these data come from, i.e., how did they get added to the knowledge base?

- What is the purpose of this rule or ruleset?
- How can you justify the use of this rule?

To answer these questions, XPL focuses on describing the inference chain leading to a conclusion, the rulesets and rules comprising that chain, and the individual premises comprising the rules. It also uses patterns (text annotations containing variables) and procedures (executable rulesets) associated with the rules and rulesets. In problem domains that have a well-understood deep model of the mechanisms underlying the surface rules, the procedures associated with rules can be used to generate sophisticated explanations of a particular rule or justifications for its use in the current context. The techniques required to generate these justifications are somewhat domain-dependent, being based upon the form and structure of the deep models involved. However, domain-independent techniques for applying additional knowledge of this type to the problem of explanation are under development (Neches et al., 1985). For problem domains with poorly formalized or no deep models, the text patterns associated with the rules can be used to produce the needed explanations.

XPL provides direct access to explanations, such as the inference chain display, as well as indirect access through special mechanisms that help the expert system builder tailor the explanation facility to his or her particular needs. For example, XPL's set of window-creating commands permits the system builder to easily define, shape, and feed explanation windows. These commands exist at the ROSIE level and may be used to create a *definitional window stack*, a mechanism for defining terms used by XPL and the performance program. The system builder creates pop-up windows to display definitions of terms the user indicates are confusing. If a definition itself contains questionable terms, the user can request that they too be defined, creating a stack of windows that can be accessed forward or backward.

ORGANIZATION OF THE REPORT

Section II of this report presents a brief introduction to the ROSIE knowledge engineering language, describing what it is and how it is used. Section III describes the features of XPL in some detail and provides examples of its use. Section IV presents our conclusions, including our evaluation of XPL and suggestions for possible extensions to the system.

II. THE ROSIE KNOWLEDGE ENGINEERING LANGUAGE

ROSIE is a general-purpose knowledge engineering language designed specifically for developing expert systems. The ROSIE language has evolved from a relatively simple initial design (Waterman et al., 1979) to a sophisticated expert system building tool (Sowizral and Kipps, 1985).

OVERVIEW OF ROSIE

The distinguishing characteristic of ROSIE is a highly readable English-like syntax, designed to facilitate the process of formalizing the expertise and integrating it into an executable program. With ROSIE, the knowledge engineer can easily translate ideas into executable code, using substantially the same terminology that the domain expert uses. ROSIE's English-like syntax also improves the interactions among those involved in the knowledge acquisition process. A well-written ROSIE program is accessible not only to the programmer or knowledge engineer, but also to the domain expert, system users, and others as well.

ROSIE has been used to support the development of many significant applications (Waterman, 1986). It brings to each application a variety of language and programming-environment features, including:

- Extended variations of the data types and control structures found in most symbolic languages.
- Rulesets to modularize and scope rules, localizing the context in which they apply.
- High-level data types for manipulating units of procedural, declarative, and descriptive knowledge as data.
- A string pattern matcher which supports advanced I/O operations.
- A demon facility to provide event-driven program control.

Such features as rulesets, demons, and the string pattern-matcher blend with the naturalness of ROSIE's syntax to produce a comfortable and expressive environment for the construction of expert rules.

ROSIE adapts to a wide variety of tasks without embodying special-purpose problem-solving techniques. It is less structured and more

flexible than many contemporary knowledge engineering tools. For example, whereas many rule-based research tools for building expert systems permit only a rigid, uniform approach to rule organization, ROSIE permits rules organized as rulesets which can be accessed as nested, recursive procedures (Waterman and Hayes-Roth, 1983). Thus, the design of ROSIE exploits and integrates many current ideas in AI research and puts substantial modeling capabilities into the hands of expert system development teams.

RULES AND RULESETS

The principal programming structures in ROSIE are *rules* and *rulesets*. Rules correspond to the executable programming statements of other languages, while rulesets correspond to subroutines. However, ROSIE rules differ significantly from the rules in traditional production system architectures, such as OPS5 (Forgy, 1981, 1982) and EMYCIN (van Melle et al., 1984). While ROSIE rules may be written in the traditional *If-Then* format, they can also be written using other, more powerful and often more appropriate control structures, as illustrated below:

```
If the ceiling at the airfield is less than 3500 feet or
   the visibility at that airfield is less than 3 miles,
   let the weather be POOR,
otherwise,
   let the weather be GOOD.

Display the ship's destination and
   assert that destination was displayed.

For each asbestosis symptom (SYMPTOM) of the plaintiff,
   check alternative_explanations for SYMPTOM and
   assert SYMPTOM was checked for alternatives.

While any strategic objective is not defended,
   keep some friendly force on alert.
```

The ROSIE programmer can control the context in which rules are executed by organizing them into rulesets. This subroutining facility distinguishes ROSIE from most other rule-oriented systems. Like subroutines in more conventional programming languages, rulesets provide a convenient way to modularize a ROSIE program into coherent procedural units, which can then be invoked in a natural and transparent way using ROSIE's English-like linguistic structures. Programmers

may define three different types of rulesets: *procedural*, *generator*, and *predicate*. Each type serves a different function, is invoked in a different way, and returns a different type of value.

A *procedural* ruleset performs some activity; it does not return a result to the rule that invoked it. For example:

```
To move a unit to a destination:
[1] If the unit is a vessel,
    deny that unit is docked at that unit's location and
    assert that unit is docked at the destination.
[2] If the unit is a battalion,
    deny that unit does control that unit's location and
    assert that unit does control the destination.
[3] Let the location of the unit be the destination.
End.
```

This ruleset updates the database when invoked by procedure calls, such as:

```
Move USS Nimitz to Pearl Harbor.
Move some friendly battalion (which is undeployed)
  to the objective.
```

Generator rulesets produce instances of a class of items. For example, if the database contains

```
PT-67 is a boat
USS Nimitz is a ship
USS Saratoga is a ship
PT-67 is docked at Pearl Harbor
USS Nimitz is docked at Pearl Harbor
USS Saratoga is docked at Pearl Harbor
```

then a generator ruleset, such as,

```
To generate a vessel at a port:
[1] Produce every boat which is docked at the port.
[2] Produce every ship which is docked at the port.
End.
```

when invoked by

```
For each vessel at Pearl Harbor,
  display that vessel.
```

causes each computable vessel to be displayed in succession, as

PT-67
USS Nimitz
USS Saratoga

Calls to generators can be made transparently and do not detract from the readability of the code.

Predicate rulesets provide a way of computing the truth or falsity of a *proposition* (i.e., a primitive sentence). When ROSIE cannot decide the truth value of a proposition from affirmed propositions in its database, it automatically invokes a corresponding predicate ruleset if one exists. An example of a predicate is:

To decide if a vessel is seaworthy:
[1] If the vessel does pass inspection, conclude true,
otherwise, if the vessel does exhibit any signs of damage,
conclude false.
End.

This ruleset could be invoked by executing the following rule:

If USS Nimitz is seaworthy,
move USS Nimitz to Pearl Harbor.

A predicate can conclude true or false, returning a boolean value to the calling form, or it can simply terminate, returning nothing and implying an indeterminate truth value. Note that references to a predicate ruleset can not be distinguished from references to relations in the database.

ROSIE also supports a specialized type of ruleset called a *demon*. Demons selectively capture control of computations (i.e., are invoked) when a particular event occurs. Once invoked, a demon can interrogate the system's state, and either allow the event to resume or release control without continuing the event. For example, the demon

Before executing to move a ship to a destination:
[1] If some vessel at the destination is equal to the ship,
send {the ship, " is already at ", the destination} and
return,
otherwise continue.
End.

would be awakened by the procedure

Move USS Nimitz to Pearl Harbor.

Demons permit the specification of event-driven program control and are useful for maintaining consistency as the database changes.

LINGUISTIC STRUCTURES

ROSIE's three principal linguistic structures are *terms*, *actions*, and *sentences*. Actions advance the flow of control; sentences describe declarative relations; and terms act as arguments to both.

Terms are ROSIE's data objects, i.e., expressions that evaluate to one or more data primitives called *elements*. Terms serve as arguments to actions and sentences, as well as to other terms. Some illustrative terms are shown in Table 1.

Table 1

EXAMPLES OF ROSIE TERMS

Type	Term
Elementary	airfield # 5 25000 DOLLARS
Arithmetic	the current DE * (the number of exposed aircraft / the total number of aircraft)
Descriptive	the blood-gas-test result of the plaintiff every friendly battalion which is undeployed
Anaphoric	that ship that symptom
Iterative	one of F-111X, F-4X or F-16X each of lung volume and vital capacity

ROSIE supports a specialized element called a *pattern* for complex I/O and string manipulation tasks. A pattern designates a virtual set of strings. Strings can be matched for inclusion in patterns. When the pattern designates a set of only one string, it can be coerced into that string and output. Examples of patterns used for both matching and formatting text are given in Table 2.

Table 2

EXAMPLES OF ROSIE PATTERNS

Patterns	Strings
String matching patterns	*Matched Strings*
{"airfield #", 0 or more digits, end}	airfield #13
{anything, "docked at", anything, end}	The ship is not docked at Newport
Text formatting patterns	*Output text*
{the ship, " does need repair"}	USS Nimitz does need repair
{"Plaintiff: ", the plaintiff}	Plaintiff: John Smith

ROSIE *actions* represent executable operations; actions can execute procedures, modify the database, or control the execution of action blocks. Examples of actions are given in Table 3.

Table 3

EXAMPLES OF ROSIE ACTIONS

Action	Effect
assert the force was given a new directive deny the plaintiff did contribute to the injury let the objective be sortie reduction	Modify the database
cmd {"Airfield: ", the airfield, cr} determine the exposure history of the plaintiff	Execute procedures
If any enemy battalion does advance toward any objective, move some friendly battalion (which is undeployed) to that objective and report 'that battalion was directed to that objective'	Conditionally execute blocks of actions
for each force (F) in the border area, advise F to 'maintain alert status' and assert F was given a new directive	Iterate over a set

Sentences specify declarative relations whose truth or falsity can be tested. Some sentence forms test the cardinality of a class, while others test the equality of data objects, as illustrated below:

Cardinality: There is more than one force which is on alert.

Equality: The type of attack aircraft is equal to F-111X.

There also exists a type of sentence called a *proposition*, whose truth or falsity can be computed by predicate rulesets. Propositions have five basic syntactic forms, each of which captures a specific class of English usage. Examples of these five basic forms plus extensions obtained by appending prepositional phrases or changing tense are shown below:

Class Membership
- Syntactic form: *term* is a *description*
- Examples: dyspnea is a symptom of asbestosis
  flogger is a type of aircraft at airfield #1

Predication
- Syntactic form: *term* is *verb*
- Examples: the border area was undefended at 0830 hours
  the radiographic evidence is contested by the defense

Predicate complement
- Syntactic form: *term* is *verb term*
- Examples: the battle is nearly finished
  USS Nimitz was rarely deployed in the Persian Gulf

Intransitive verbs
- Syntactic form: *term* does *verb*
- Examples: the ship does float
  battalion #4 did proceed to the objective

Transitive verbs
- Syntactic form: *term* does *verb term*
- Examples: the plaintiff does have a "history of exposure to asbestos"
  the force did receive the message at 1500 hours

The ROSIE environment provides the necessary support for designing and implementing expert systems. XPL greatly enhances the usefulness of that environment.

III. THE ROSIE EXPLANATION FACILITY

XPL, the ROSIE explanation facility, is a set of mechanisms that modify and enhance the standard ROSIE 2.4 programming environment. This set extends the grammar of the ROSIE language, adds Interlisp software to support explanation, and adds ROSIE software packages that are compatible with the standard ROSIE syntax. As indicated in Fig. 1, XPL comprises a history mechanism, grammar enhancements, a backtracking mechanism, a menu package, and a win-

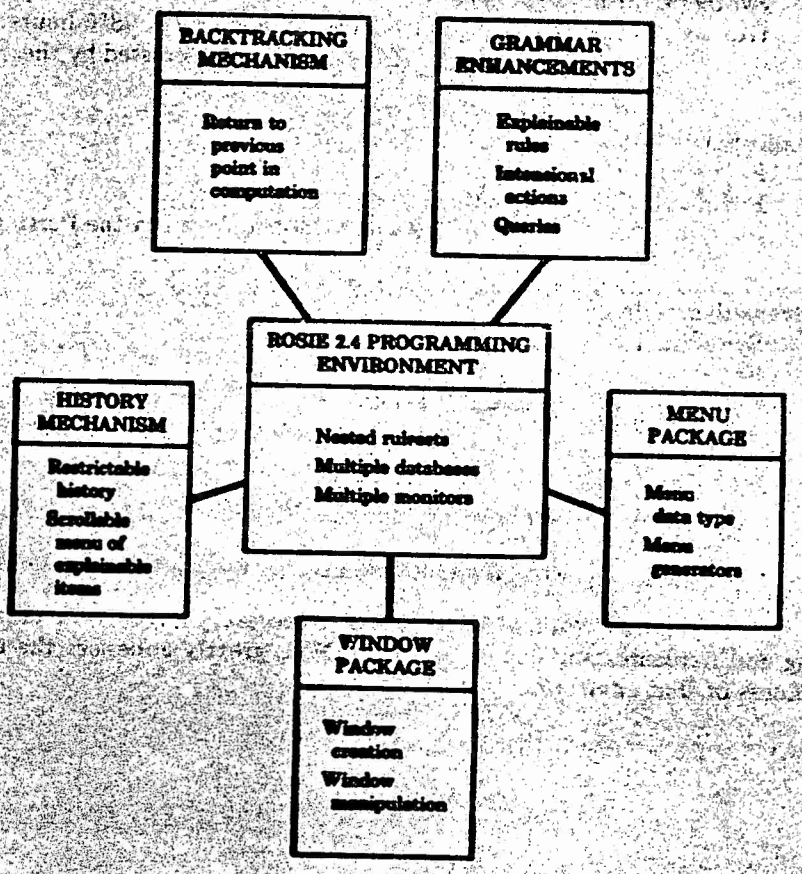


Fig. 1—Components of XPL

dow package. The examples of XPL features presented below are drawn from military and legal problem domains (i.e., tactical air targeting, product liability claim settlement, and estate planning).

HISTORY MECHANISM

XPL's history mechanism automatically records and organizes the information about a computation (action or conclusion) and stores it on a history list. The expert system developer can select in advance those computations that will be of interest to the end user, and only information about those computations will be stored. In effect, the system developer tags particular rules and rulesets so that when they are executed, records are placed on a history list. These records include (1) static and dynamic textual information, i.e., plain text and text mixed with ROSIE terms that are evaluated at run time; (2) information concerning the *local* context of the event, i.e., the assertions that satisfied the condition of a rule; and (3) information concerning the *global* context of the event, i.e., what rulesets were active at the time of the event.

XPL also provides a menu-driven interface to the history list by which the user may determine which items should be explained. These items are the set of intermediate and final conclusions made and actions taken by the expert system during execution. After the system finishes its computation, the items appear in a special overlay window and can be selected by scrolling through the list and pointing to the desired items. Once an item is selected, the user may exercise any of nine options regarding explanation (selected from a pop-up *explanation menu* window), which include both moving along and displaying the inference chain all the way back to the input data. Figure 2 shows an example of these two windows.

HISTORY LIST WINDOW	
1.	The plaintiff is JOHN SMITH.
2.	JOHN SMITH does possess complete evidence of asbestosis.
3.	The injury-value of the case is 70000 DOLLARS.
4.	The exposure-history of JOHN SMITH is SEVERE-EXPOSURE.
5.	AGE is important.
6.	The value of the case is 116000 DOLLARS.

EXPLANATION WINDOW	
1.	Description
2.	Example
3.	Premises
4.	Rule
5.	Ruleset
6.	Summary
7.	Tutorial
8.	Why?
9.	RETURN

Fig. 2—Selection windows for explanation

Whenever an item is selected, XPL attempts to display an explanation associated with it. If a *general* description—i.e., a high-level description that characterizes the item's use independent of the current context—is available, it is displayed. If a general description is not available, XPL displays an *example* explanation consisting of one or more specific examples of how the item was used in the current context. If neither of these is available and the item is a rule, the *instantiated premises* (the subset of premises in the rule that caused it to fire in the current context) are shown.

The user selects the additional type of explanation desired by pointing to an *explanation menu item*. The available options are summarized in Table 4.

The user can select *Descrip...* to display a high-level explanation of why a particular conclusion was reached. *Example* can be selected repeatedly to provide different specific examples showing why or how the conclusion was reached. The user can select *Premises* to see the precise form of the instantiated premises from the rule that produced the conclusion. *Rule* or *Ruleset* may be selected to see the actual definition of the rule or ruleset involved.

The *Summary* option provides a high-level explanation of the ruleset that led to the concluded item. *Tutorial* provides access to tutorial text illustrating the concepts embodied in the ruleset. Upon selection of this option, the first piece of tutorial text is displayed, and the explanation menu appears, as in Fig. 3. *Tutorial* can then be selected repeatedly to cycle through the available text.

EXPLANATION WINDOW	
1.	Tutorial
2.	Summary
3.	Ruleset
4.	RETURN

Fig. 3—Tutorial explanation window

The *Why?* option allows the user to follow the logical chain of reasoning the system used to arrive at a particular conclusion. Each time *Why?* is selected, the system moves one step up the inference chain, displaying the explanation associated with the rule that invoked the previous ruleset. At each step, the user may exercise any of the options described in Table 4.

Table 4

AVAILABLE EXPLANATION OPTIONS

Selection	Result
Description	The system presents a high-level explanation describing why the conclusion was reached.
Example	The system gives specific examples illustrating why the conclusion was reached.
Premises	The system displays an instantiated form of the premises from the rule used to reach this conclusion.
Rule	The system displays the uninstantiated rule used to reach this conclusion.
Ruleset	The system displays the entire ruleset containing the rule used to reach this conclusion.
Summary	The system presents a high-level explanation of the ruleset used to reach this conclusion.
Tutorial	The system gives information illustrating the concepts embodied in the ruleset used to reach this conclusion.
Why?	The system explains other rules in the inference chain for the conclusion.
RETURN	The user exits the explanation mod

The use of the *Why?* option is illustrated by the following example, in which the user has indicated that the item of interest on the XPL history list is the plaintiff's history of exposure to asbestos. The user wants to trace back through the inferences leading to the conclusion that the plaintiff does have a history of exposure to asbestos:

I did conclude that JOHN SMITH does have
"a history of exposure to asbestos"

in To decide A PLAINTIFF does have A
HISTORY-OF-EXPOSURE-TO-ASBESTOS rule 2

because:

By history of exposure we mean that the plaintiff experienced either a severe or significant exposure to asbestos.

> Why? ***[were you trying to determine that the plaintiff had a history of exposure to asbestos]***

LEVEL 2

I needed the conclusion that
'JOHN SMITH does have "a history of exposure to asbestos"'
in order to conclude that
'JOHN SMITH does possess "complete evidence of asbestosis"'

in To DETERMINE-DOCUMENTATION-OF-INJURY rule 2

because:

We consider evidence to be complete if the plaintiff has a history of exposure to asbestos, symptoms of asbestosis, and both laboratory evidence and radiographic evidence of parenchymal asbestosis.

The user can select these explanation options either after processing is completed or during processing. To select them during processing, the user interrupts the program, invokes the history list window mechanism, and then selects the desired explanation options. For example, the user could trace back through the logical chain of reasoning used by the system to find out why a particular question was asked or why some intermediate conclusion was reached.

GRAMMAR ENHANCEMENTS

The three main enhancements to the ROSIE grammar are the *explainable rule capability*, a ROSIE data type called an *intensional procedure*, and the *query capability*, which consists of special grammar constructs for defining queries directed to the user.

Explainable Rules. Rules and rulesets are tagged for future explanation and placement on the history list by associating them with special grammatical forms. Only tagged rules and rulesets participate in the explanation processes; non-tagged constructs will not appear in the inference chain, nor will their computations be available for explication. The special grammatical forms also can associate each rule or ruleset with either (1) fixed text, (2) text with variables (ROSIE pat-

terms), or (3) arbitrary procedures, i.e., ROSIE rulesets designed to dynamically create an explanation based on the current context. In the following discussion, we refer to these associated explanations simply as explanations.

The specification of an explanation has the same basic form regardless of the item that is to be explained. The specification may contain one *general* explanation and/or any number of *example* explanations, as shown below:

... <*conclusion*> because, in general, <*general explanation*>;
for example, <*specific explanation*>
and <*specific explanation*>
and <*specific explanation*>.

The *general* and *example* explanations may be either strings of text enclosed in double quotes, ROSIE patterns containing text and terms evaluated at run time when the explanation record is created, or ROSIE patterns specifying a call to an arbitrary ruleset. Examples of each type of explanation are given in Table 5.

The *general* and *example* explanations may also be ROSIE tuples that could be used to implement a more general explanation facility. If tuples are used, however, the expert system builder must customize certain rulesets in the menu package to access the information contained in the tuples. Tuples may be used to permit the system designer to create different levels of explanation for users with different levels of knowledge. For example, the expert system builder could use XPL to create a three-level explanation system for novice users, expert users, and system builders. Each explanation form could be a 3-tuple

Table 5

TYPES OF EXPLANATIONS

Type	Form
Text	"Forces are deployed on a demand basis. A percentage of forces will be held in reserve to support priority confrontations."
Pattern	{"Since ", the plaintiff, " has only ", the plaintiff's exposure to asbestos, ", we conclude there is no history ", "of asbestos exposure."}.
Ruleset	{the explanation for force deployment}.

containing the three different explanation texts for the different types of users. Another more interesting approach to providing multilevel explanations is based on 2-tuples, in which the first item is a basic explanation and the second is a call to a procedure specifying the transformations needed to create each of the desired levels of explanation. Another use of tuples in explanation is presented in the discussion of the ROSIE menu package on pp. 26–27.

Explanations may be integrated into a ROSIE program through their association with any of the following ROSIE constructs:

- A procedural, generator, or predicate ruleset
- An assertion into the database
- The conclusion from a predicate ruleset
- The value produced by a generator ruleset
- The invocation of a procedural ruleset

The first construct is a special, independent "explain" rule inserted at a ruleset's beginning to explain the ruleset's actions. The others are "because" clauses added to a rule to explain what it means when the rule's premise is true. These also serve as tags to indicate when explanations should be added to the history list. The use of explanation with each of these constructs is discussed and illustrated below. In these examples, XPL extensions to ROSIE are shown in italics.

An explanation may be associated with any ROSIE ruleset (procedural, generator, or predicate). The specification of that explanation immediately follows the ruleset's header, as shown in Fig. 4.

```
To generate a percentage-reduction to aircraft at an airfield after
     an attack on a target:
[1] EXPLANATION: because, in general, {"The airfield selection rules ",
     "calculate the sortie reduction that would result from attacking ",
     "each recommended target element separately and then combines ",
     "these reductions to determine the effect of attacking groups of ",
     "elements."},
          for example, {"We are currently calculating the ",
     "percentage reduction to aircraft at ", the airfield,
     " after an attack on ", the target, "."}.
.
.
.
End.
```

Fig. 4—Example of associating an explanation with a ruleset

The *general* and *example* explanations for rulesets are accessed via the *Summary* and *Tutorial* options from the explanation menu, respectively. They provide high-level descriptions of the operation and intent of all the rules acting as a single unit. The *general* and *example* explanations for individual rules are accessed via the *Description* and *Example* options.

An explanation may be associated with an assertion into the database, as shown in Fig. 5. In this case, the "because" clause triggers the explanation's inclusion in the history list when the conclusion is asserted.

To determine-documentation-of-injury:

.
.
.

[2] If the plaintiff does have (a history of exposure to asbestos) and
the plaintiff does manifest radiographic-evidence of asbestosis and
the plaintiff does have dyspnea as a symptom and
(the plaintiff does have laboratory-evidence of asbestosis or
the plaintiff does demonstrate symptoms of asbestosis),
conclude that the plaintiff does possess
"complete evidence of asbestosis" because, in general,
{*"We consider evidence to be complete if the plaintiff has ",*
"a history of exposure to asbestos, symptoms of asbestosis ",
"and both laboratory evidence and radiographic evidence of ",
"parenchymal asbestosis."}.

.
.
.

End.

Fig. 5—Example of associating an explanation with an assertion into the database

An explanation may be associated with the conclusion from a predicate ruleset, as shown in Fig. 6. In this example, the predicate returns "TRUE" and is recorded with the *general* and *example* explanations given.

An explanation may also be associated with the value produced by a generator, as shown in Fig. 7. Here, the value produced is the instantiation of "that battalion".

```
To decide if a weapon-system does satisfy policy-rules for use against
      a target at an airfield:
   .
   .
   .
[2] If the weapon-system does fail to satisfy (some ROE for
                the airfield),
    conclude that the weapon-system does not satisfy policy-rules for
        use (against the target) at the airfield is TRUE
            because, in general, {"Before a strike can be made on ",
                "a target element at an airfield, that strike must ",
                "satisfy rules of engagement at that airfield."},
            for example, {"A strike at ", the airfield, " with ",
                the weapon-system, " fails to satisfy the ROE: ",
                that ROE}.
   .
   .
   .
End.
```

Fig. 6—Example of associating an explanation with the conclusion from a predicate ruleset

```
To generate a battalion in a sector:
   .
   .
   .
[2] For each battalion which is stationed in the sector,
    produce that battalion as the battalion in that sector
        because, in general, {"Any battalion stationed within ",
            "a sector can be called upon to defend objectives ",
            "of that sector."}.
End.
```

Fig. 7—Example of associating an explanation with the value produced by a generator

An explanation may be associated with the invocation of a procedural ruleset, as shown in Fig. 8. Here, the procedure being invoked is "deploy-forces" and the explanation for the procedure's invocation is triggered by the "because."

```
"Associating an explanation with the invocation" "of a procedural ruleset."

To enact defense scenario:
[3] Do deploy-forces because, in general, {"Force must be deployed
                against advances by ", the enemy force, " in order to ",
                the current goal for deploy-forces, "."}
End.
```

Fig. 8—Example of associating an explanation with the invocation of a procedural ruleset

Intensional Procedures. The intensional procedure is a ROSIE element that allows the knowledge engineer to store an unexecuted action in the database for later execution. This procedure defers the execution of the verb in the command statement. Intensional procedures can be used for implementing framelike capabilities in ROSIE. For example, in a frame-based system, it is necessary to be able to define an unexecuted ruleset as the value of a frame slot and later execute the ruleset in a new environment. This can be done in the explainable version of ROSIE by defining a ruleset called "Put VALUE for a SLOT of a FRAME," which sets the value of a given slot in a given frame, and a ruleset called "Invoke VALUE for a slot of a FRAME," which executes the value in that slot. Examples of these two rulesets are shown below:

```
To put a value for a slot of a frame:
[1] Let the filler (for the slot) of the frame be the value.
End.

Invoke ROE of AIR STRIKES.

To invoke a slot of a frame:
[1] Execute (the filler) (for the slot) of the frame.
End.
```

Assume that one wants to add the following unexecuted ruleset to the frame AIR STRIKES under the slot ROE (rules of engagement):

To display-rules-of-engagement against a hostile among non-belig into a window:

[1] Output {"A 25NM buffer zone established within the hostile country " the hostile, " along its borders with the non-belligerent ", "nations ", the non-belig, ". Attack of targets within this", " buffer zone is prohibited."} to the window.

.
.
.

End.

To add this ruleset to the frame AIR STRIKES under the slot ROE, execute the command:

Put 'display-rules-of-engagement against 'the hostile country' among 'the non-belligerent-countries' into 'the window"
for ROE
of AIR STRIKES.

The knowledge engineer could later use the *Invoke* command to execute this ruleset, generating an explanation of the applicability of air strikes in accordance with the rules of engagement. For example, if the knowledge engineer issued the command:

Invoke (the slot-value for ROE of AIRSTRIKES).

The system would reply:

A 25NM buffer zone established within the hostile country REDLAND along its borders with the non-belligerent nations OILLAND and CAMELLAND. Attack of targets within this buffer zone is prohibited.

.
.
.

Queries. XPL provides special grammar constructs for defining queries directed to the user. For "yes/no" questions, the system builder can easily specify the explanations to be displayed when the user is being queried. In this case, XPL displays the question and then presents the user with the following menu:

1. YES
2. NO
3. WHY?
4. ELABORATE!
5. PAUSE

The user has the option of either answering the question (*Yes* or *No*), accessing special explanations provided by the system builder (*Why?* and *Elaborate!*), or invoking the ROSIE break package to debug the system (*Pause*). The ruleset below illustrates the format for specifying the explanations. Note that the explanation associated with a *conclude* clause will be attached to that conclusion's history list record for later access via the explanation menu.

```
To decide if a client is married:
    [1] Question: {"Is ",the client," married?  "},
        If yes: conclude that the client is married
                because, in general,
                    "This is what you told me.";
        If no: conclude that the client is not married
               because, in general,
                   "This is what you told me.";
        If why: because, in general,
                    {"The rules for determining the ownership of ",
                    "property are different for married clients."};
        Elaboration: for example,
                    {"For example, only married couples may hold ",
                    "property as community property."}.
    End.
```

Again, the explanations following the yes, no, why, and elaborate portions of the rule may be in the form of text, patterns with variables, or calls to arbitrary rulesets.

For more general questions, XPL provides grammar constructs similar to those just described. The system builder specifies an input pattern against which the user's input is matched. The user may type *why* or *elaborate* to access the same types of explanations as specified above. The rule shown below illustrates the format required to specify the explanations:

```
[1] Question: {"What is ", the battalion, " attrition rate?"},
      Read: {".", 0 or more digits (bind the battalion's
                attrition-rate to a number), cr};
   Default: conclude the battalion's attrition-rate is .18
      because, in general, {"An attrition of .18 ",
         "can be expected on the average for defending ",
         "forces."};
   If why: because, in general, {"A battalion's attrition rate ",
      "decides the readiness of the force."};
   Elaboration: in general, {"Input the attrition as a real ",
         "number between 0 and 1 representing the ",
         "fractional attrition rate to force."},
      for example, {"The default attrition is .18."}.
```

BACKTRACKING MECHANISM

XPL provides a *catch and throw* mechanism, which allows the system developer to write code that can explore a chronological line of reasoning and then back up to an earlier point in that line if the reasoning proves to be unsatisfactory. This is accomplished through a new monitor mechanism which allows a ruleset to be restarted after processing has reached an arbitrary depth.

The catch and throw mechanism is particularly valuable for expert systems that make a series of initial assumptions about the problem domain and the particular problem being solved. Making such assumptions can speed system execution by obviating the need to ask the user a lot of unnecessary questions. However, during the course of the interaction, the expert system may determine that particular assumptions were incorrect. If this happens, the system must remove certain conclusions, contradict the incorrect assumptions, backtrack to the appropriate point in the program and restart the computation.

To define the backtracking departure and destination points, the knowledge engineer adds the statement

Execute with marker <name of marker>

to the destination ruleset, and the statement

Return to marker <name of marker>

to the departure ruleset, as illustrated in Fig. 9.

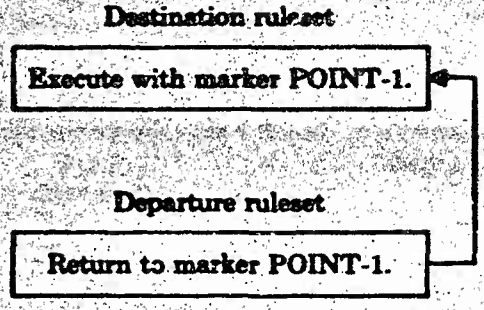


Fig. 9—Backtracking in ROSIE

For example, assume that a military planning expert system must backtrack to the start of a ruleset for establishing force deployment criteria and that this backup occurs when certain assumptions are not valid. The ruleset could have the following form:

To establish criterion for deploying-forces:
Execute with marker FORCE DEPLOYMENT.
[1] Unless there is no vulnerable supply-line,
determine-supply-line-requirements.

.
.
.

End.

The system must also contain a ruleset that determines the validity of the assumptions and initiates backtracking when those assumptions are invalid. In addition, it must correct the invalid assumptions and any conclusions that depended upon those assumptions. For example, the system could contain the following rule:

If the user does not want-the-assumptions,
return to marker FORCE DEPLOYMENT.

Here, the user must decide which assumptions are not valid. Once this is done, the invalid assumptions are removed (by code explicitly provided by the expert system builder), and control returns to the destination ruleset. This *catch and throw* backtracking mechanism is especially useful for planning systems, where assumptions are constantly being made and revised.

MENU PACKAGE

The XPL menu package effectively implements the Interlisp-D menu data type in ROSIE. A menu is a ROSIE tuple, each element of which is one menu item. Each item may be any ROSIE data object, including a tuple itself.

Two generators allow ROSIE to interface with the user through a menu, as illustrated in Table 6. The first allows the user to select exactly one item from a menu; the second permits multiple selections or no selection at all.

Table 6

MENU PACKAGE GENERATORS

Generator	Function
SINGLE-ITEM from MENU	Displays the menu and returns a value as soon as the user selects one
SELECTED-ITEM from MENU	Displays the menu and returns all items selected by the user

The following example illustrates the use of these generators to produce a menu interaction. First, the menu items must be defined, e.g.,

Let the trade-menu be <shipbuilder, carpenter, insulator>.

Let the symptom-menu be <"Dyspnea", "Cur Pulmonale", "Cyanosis", "Clubbing of Fingers">.

When the expert system activates one of the generators, a menu is displayed and the item selected is returned as the generator's value. For example,

Let the plaintiff's trade be the single-item
from the trade-menu.

creates a display with the following visual representation,

1. SHIPBUILDER
2. CARPENTER
3. INSULATOR

and assigns the selected value as the plaintiff's trade. Similarly, the following generator call,

> Assert every *selected-item from the symptom-menu*
> is a symptom of the plaintiff.

creates the following visual display,

1. Dyspnea
2. Cur Pulmonale
3. Cyanosis
4. Clubbing of Fingers

---ENTER---

and defines every selected value to be a symptom of the plaintiff.

Explanations can be attached to the menu items by representing them as tuples, where the first tuple element is assumed to be the display form (i.e., the form that will be visually displayed in the menu); the second tuple element is the return value (i.e., the data object that will be returned when this item is selected in the menu); and the third tuple element is the explanation or elaboration made available to the user if he or she is unsure of the meaning of the item. The elaboration is optional. If the return value is missing, the display form is returned. Thus, the menu item,

<"installed insulation", INSULATOR, {the elaboration for insulator}>

causes the item *installed insulation* to be added to the menu of items presented to the user for selection. If the user selects *installed insulation* from the menu, the value *INSULATOR* is returned as the user's response. If the user indicates that he or she wants an elaboration of the term *installed insulation*, the explanation generated by the ruleset called "*the elaboration for insulator*" is displayed.

XPL was developed on Xerox 1100 series computers, so the menu package was designed to use the Interlisp-D menu data type which is activated by a mouse-type control. Some development was also done on VAX computers, however, so a keyboard-activated version is available which is transparent to the expert system builder. With this version, the user simply enters the number(s) of the desired item(s), followed by a carriage return.

WINDOW PACKAGE

The XPL window package takes advantage of the windowing capabilities of the Xerox 1100 series computers. This package allows the system builder to create and manipulate windows in much the same way any other ROSIE data object is manipulated. The following example illustrates the creation and initialization of a window:

```
Assert testwindow is a window and
let testwindow's region be <500, 30, 500, 250> and
    testwindow's title be "The test window" and
    testwindow's font be <GACHA, 12, BOLD> and
    testwindow's titlefont be <TIMESROMAN, 24> and
    change pagehold of testwindow to DONT and
    clear out testwindow.
```

This ROSIE command creates a window named testwindow; gives it the indicated region, title, font and titlefont; keeps it from pausing when filling with a page of text; and finally ensures that printing will start at the upper left corner of the window.

Windows may be opened, closed, cleared, and written into by using special ROSIE procedures. A formatting procedure is also available which ensures that lines of text are terminated at the end of a word rather than at an arbitrary point.

Here again, for purposes of portability and to support development using VAX computers, a version of the package is available which collapses the output of all window commands to the standard dumb terminal screen and provides word-wrapping of text lines.

IV. CONCLUSIONS

The XPL facility provides ROSIE with a mechanism and framework that the knowledge engineer can use to build expert systems that explain their operation. XPL gives the user a basic explanation package, i.e., the ability to move either backward or forward through the inference chain to see how a conclusion was derived, and it provides a set of "hooks," or software tools, the system builder can use to customize the explanation package to the particular needs of the application. The facility makes especially good use of the windowing capabilities of the host computer, in this case a Xerox 1100 series machine.

The selectable paraphrasing technique, in which particular rules and procedures are defined by the expert system builder to be explainable, appears to solve a problem that is experienced in many general-purpose knowledge engineering languages, the problem of structural complexity: With a language that contains not only *If-Then* rules, but also many conventional programming language constructs, such as for-loops and recursive procedure calls, how can the operation of an application program be explained? Our solution involves (1) marking and saving a historical trace of the rules and procedures deemed important to the user, and (2) restricting constructs that can be marked to a set of standard, relatively simple syntactical forms. The main disadvantage of this approach is that it may preclude the use of some esoteric but possibly more compact syntactic forms available in the language. Thus the inclusion of an explanation facility enforces a particular programming style on the system builder, a limitation that is seldom clearly recognized or understood.

A number of extensions would benefit XPL, including graphics, monitoring, and editing features. The facility needs a sophisticated graphics capability for displaying the current state of the computation. This could be at ıed by a floating window that would be able to traverse the entire ınference tree and display portions of the inference chain graphically. Another possibility would be a zoom window that could display the entire tree at a very abstract level of detail and then focus on specific portions showing more or all of the detail.

XPL would also be enhanced by automatic "explanation monitoring," that is, a menu of specialized windows that continually answer particular questions. Explanation monitoring would make it easy for the system builder to define the window, question, and monitoring capabilities. Useful monitoring windows would include a status window

that answers the question, "What are you doing now?"; a graphical display window that answers the question, "Where are you in the calculation?"; and an explanation window that answers the question, "How did you arrive at that conclusion?"

Finally, XPL could benefit from a special refinement mechanism that would allow a user to change a fact or edit a rule displayed during an explanation and then rerun the expert system, with minimum obtrusiveness. This would require a dependency structure to determine what was still relevant after the changes were made, so the computation and user queries would not have to start over from the beginning. Such a mechanism would be extremely useful for refining a fledgling knowledge base and for sensitivity analyses of particular applications.

BIBLIOGRAPHY

Anderson, R. H., Margaret Gallegos, J. J. Gillogly, R. Greenberg, and R. Villanueva, *RITA Reference Manual*, The Rand Corporation, R-1808-ARPA, September 1977.

Anderson, R. H., and J. J. Gillogly, *Rand Intelligent Terminal Agent (RITA): Design Philosophy*, The Rand Corporation, R-1809-ARPA, February 1976.

Callero, Monti, Donald A. Waterman, and James R. Kipps, *TATR: A Prototype Expert System for Tactical Air Targeting*, The Rand Corporation, R-3096-ARPA, August 1984.

Callero, Monti, Lewis Jamison, and D. A. Waterman, *TATR: An Expert Aid for Tactical Air Targeting*, The Rand Corporation, N-1796-ARPA, January 1982.

Clancey, W. J., "Extensions to Rules for Explanation and Tutoring," in B. Buchanan and E. Shortliffe (eds.), *Rule-Based Expert Systems*, Addison-Wesley, Reading, Mass., 1984.

D'Ambrosio, B., "Building Expert Systems with M.1," *BYTE*, Vol. 10, No. 6, June, 1985, pp. 371–375.

Fain, J., F. Hayes-Roth, H. Sowizral, and D. Waterman, *Programming in ROSIE: An Introduction by Means of Examples*, The Rand Corporation, N-1646-ARPA, February 1982.

Fain, J., D. Gorlin, F. Hayes-Roth, S. Rosenschein, H. Sowizral, and D. Waterman, *The ROSIE Language Reference Manual*, The Rand Corporation, N-1647-ARPA, December 1981.

Forgy, C. L., *OPS5 User's Manual*, Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, July 1981.

———, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, 1982, pp. 17–37.

Klahr, Philip, and Donald A. Waterman (eds.), *Expert Systems: Techniques, Tools, and Applications*, Addison-Wesley, Reading, Mass., 1986.

Neches, R., W. R. Swartout, and J. Moore, "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985.

Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier Scientific Publishing Company, Amsterdam, 1976.

Sowizral, H. A., and J. R. Kipps, *ROSIE: A Programming Environment for Expert Systems*, The Rand Corporation, R-3246-ARPA, October 1985.

Taylor, E. C., "Developing a Knowledge Engineering Capability in the TRW Defense Systems Group," *The AI Magazine*, Summer 1985, pp. 58–63.

van Melle, W., E. H. Shortliffe, and B. G. Buchanan, "EMYCIN: A Knowledge Engineer's Tool for Constructing Rule-Based Expert Systems," in B. Buchanan and E. Shortliffe (eds.), *Rule-Based Expert Systems*, Addison-Wesley, Reading, Mass., 1984, pp. 302–328.

Waterman, D. A., *A Guide to Expert Systems*, Addison-Wesley, Reading, Mass., 1986.

Waterman, D. A., R. Anderson, F. Hayes-Roth, P. Klahr, G. Martins, and S. Rosenschein, *Design of a Rule-Oriented System for Implementing Expertise*, The Rand Corporation, N-1158-ARPA, 1979.

Waterman, D. A., and F. Hayes-Roth, "An Investigation of Tools for Building Expert Systems," in R. Hayes-Roth, D. A. Waterman, and D. Lenat (eds.), *Building Expert Systems*, Addison-Wesley, Reading, Mass., 1983.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: R-3406-ARPA/RC
2. GOVT ACCESSION NO.: AD-A174503
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): An Explanation Facility for the Rosie Knowledge Engineering Language
5. TYPE OF REPORT & PERIOD COVERED: Interim
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Donald A. Waterman, Jody Paul, Bruce Florman, James A. Kipps
8. CONTRACT OR GRANT NUMBER(s): MDA903-85-C-0030

9. PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, CA 90406
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, Department of Defense, Arlington, VA 22209
12. REPORT DATE: September 1986
13. NUMBER OF PAGES: 32

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report): Approved for Public Release: Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): No Restrictions

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): Artificial Intelligence, Programming Languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number): See Reverse Side

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This report describes the design and implementation of an explanation facility for the ROSIE knowledge engineering language, an expert system building environment developed at The Rand Corporation. An explanation facility is critical in the construction and utilization of an expert system, since it provides the mechanisms by which the expert system can explain its performance. These mechanisms typically show the user (1) why and how the expert system reached a given conclusion, (2) why the system asked for particular information, (3) how rules and data got added to the knowledge base, (4) what is the purpose of a rule or ruleset, and (5) why the use of a rule is justified.

END

DATE

FILMED

1-87

UNCLASSIFIED

PLEASE DO NOT RETURN
THIS DOCUMENT TO DTIC

EACH ACTIVITY IS RESPONSIBLE FOR DESTRUCTION OF THIS DOCUMENT ACCORDING TO APPLICABLE REGULATIONS.

PROPERTY OF DACS

UNCLASSIFIED