ARPA ORDER NO.: 3473
8P10 Information Processing Techniques

R-2171-ARPA
February 1978

Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition

D. A. Waterman

A Report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA-903-78-C-0029.

Library of Congress Cataloging in Publication Data

Waterman, Donald Arthur, 1936-
Rule-directed interactive transaction agents.

Bibliography: p.
1. Interactive computer systems. I. Title.
QA76.9.I58W37 001.6'44'04 77-28172

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER: R-2171-ARPA (14) RAND/R-2171-ARPA	3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle): (6) Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition,	5. TYPE OF REPORT & PERIOD COVERED: Interim rept'y
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR: D. A. Waterman (10) Donald Arthur/Waterman	8. CONTRACT OR GRANT NUMBER: DAHC15-73-C-0181 ARPA Order 3041
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: Feb[illegible]78 (11)
	13. NUMBER OF PAGES: 53 (13) 61 pp.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (15) DAHC15-73-C-0181, MDA-903-78-C-0029	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

DDC RECEIVED APR 19 1978 F

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

RITA
Computer Programs
Artificial Intelligence
Data Processing Terminals
Heuristic Methods
Adaptive Systems

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

see reverse side

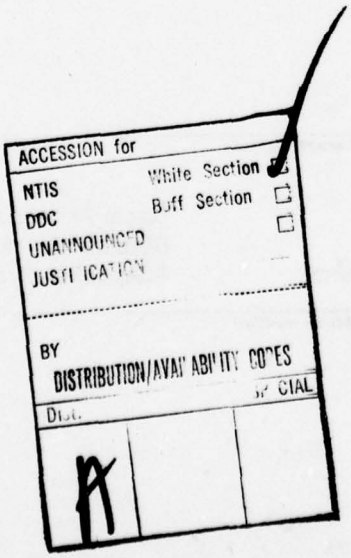
296 600

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A description of computer programs, called *user agents*, that can perform various tasks for the user. Interacting with users, these agents can learn facts and store them in a data base or learn data-manipulation procedures and represent them as programs. The agents are written in RITA, the Rule-directed Interactive Transaction Agent system, and are organized as sets of IF-THEN rules. Four types of RITA agents are described: (1) an exemplary programming agent that "watches" a user perform a task, then writes a program to perform the same task; (2) a self-modifying agent that performs file transfer tasks, modifying itself to perform them with less help each time; (3) a tutoring agent that watches demonstrations of interactive computer languages or local operating systems, then creates teaching agents for assisting new users; (4) a reactive-message creating agent that elicits text to create a reactive message, which is then sent to a recipient who interacts with it, automatically returning a response to the sender. (author)



ARPA ORDER NO.: 3473
8P10 Information Processing Techniques

R-2171-ARPA
February 1978

Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition

D. A. Waterman

A Report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

LPH—DARPA

Rand
SANTA MONICA, CA. 90406

PREFACE

This report describes results achieved by applying current production system technology, as embodied in the Rule-directed Interactive Transaction Agent (RITA) system, to the problem of developing computer programs that are capable of learning new facts and procedures by observing and interacting with a user. RITA was developed under the sponsorship of the Information Processing Techniques Office of the Defense Advanced Research Projects Agency (ARPA) for use as a front end to remote computing systems, and as a limited heuristic modeling tool. Here its use as a man-machine interface that can acquire or learn new information is explored in depth.

The report documents the current status of a Rand effort to develop sophisticated information processing mechanisms for the acquisition, organization, and utilization of knowledge that should help users interface comfortably with complex computer operating systems. The work described here, representing a first step in the development of knowledge acquisition techniques, should be of interest to computer and information scientists involved in matching the needs and abilities of computer users with the characteristics of computer systems they would like to access. It should also be of interest to researchers studying the development of learning programs and program synthesis.

SUMMARY

This report describes the development of computer programs called *user agents*, which, through interaction with users, can either learn new facts and store them in a data base or learn new procedures for data manipulation and represent them as programs. These programs are written in RITA, the Rule-directed Interactive Transaction Agent system, and are organized as sets of IF-THEN rules or "production systems." The programs are able to act as "personal computer agents" to perform a variety of tasks.

Four types of RITA agents are described:

1. An exemplary programming agent that watches a user perform an arbitrary series of operations on the computer and then writes a program (a new RITA agent) to perform the same task.
2. A self-modifying agent that performs ARPAnet file-transfer tasks for the user, modifying itself so that each subsequent time it performs the task, it can do so with less help from the user.
3. A tutoring agent that watches an expert demonstrate the use of an interactive computer language or local operating system and then creates a teaching agent that can help naive users become familiar with the language or system demonstrated by the expert.
4. A reactive-message creating agent that elicits text from a user (the sender) and uses it to create a new RITA agent which is a reactive message; the reactive message is then sent to some other user (the recipient), who interacts with it. During the course of the interaction, a record of the recipient's responses is sent back to the sender.

ACKNOWLEDGMENTS

The comments and criticisms of Robert H. Anderson and other staff members of the Information Sciences Department at The Rand Corporation are gratefully acknowledged.

CONTENTS

I. INTRODUCTION

RITA is a specialized software system that combines production-system control structure with man-machine design technology (Anderson and Gillogly, 1976a). The goal of the RITA design and development effort is to produce a system that can ultimately reside in a computer terminal and execute programs, called *agents*, which provide an intelligent interface between the user and the outside computer world. The RITA architecture is interesting and useful for three reasons: First, the use of a production-system control structure provides the degree of simplicity and modularity needed to make program organization straightforward and program modification relatively easy. Second, the system is human-engineered in the sense that the programs, or RITA agents, have an English-like syntax which makes them easy to write and understand. Thus it is possible to create RITA agents that are self-documenting. Finally, language primitives in RITA permit the user to interact with other computer systems, even to the extent of initiating and monitoring several jobs in parallel on external systems.

RITA is the embodiment of a particular production-system architecture; that is, within the RITA system one can create specific production systems or RITA agents. Production systems have a long and diverse history, originating from early work in symbolic logic (Post, 1943). Current production systems can be thought of as a generalization of Markov normal algorithms (Markov, 1954; Galler and Perlis, 1970). A production system is a collection of rules of the form *conditions* → *actions* (Newell and Simon, 1972; Waterman, 1976b; Waterman and Hayes-Roth, 1978), where the conditions are statements about the contents of a global data base and the actions are procedures that may alter the contents of that data base. Many types of production system architectures have been developed in the past few years. Some are designed to facilitate adaptive behavior (Waterman, 1970, 1975, 1976a), some to model human cognition and memory (Newell, 1972, 1973; Newell and McDermott, 1974), and others to study the usefulness of production-system control structures in artificial-intelligence tasks (Buchanan and Sridharan, 1973; Lenat, 1976; Rychener, 1975, 1976; Vere, 1975). What these architectures have in common is that they are all left-hand-side (LHS) or condition-driven, i.e., when all the conditions of a production rule are true with respect to the data base, the rule "fires," causing the associated actions to be executed. Another type of production-system architecture currently in use is based on a right-hand-side (RHS) or action-driven* control structure (Anderson and Gillogly, 1976a; Davis, et al., 1975; Davis, 1976; Duda, Hart, and Nilsson, 1976; Klahr, 1975; Shortliffe, 1976). Here, the system is given a condition to make true, a premise to prove, or, in effect, a question to answer through deductive inference. The right-hand sides of rules are examined to find one which, when executed, will make the desired condition true or prove the premise being deduced. When such a rule is found, its left-hand side is examined to see if all its conditions are true. If they are, the rule is fired; if not, the process continues recursively in an attempt to make each condition in the left-hand side of the rule true.

The RITA architecture encompasses both LHS and RHS control schemas; thus,

*Also referred to as goal-driven, consequent-driven, or backward-chaining.

it is possible to create RITA agents which are entirely LHS-driven, entirely RHS-driven, or some combination of both. The production rules accessed from the LHS are called *rules*, and those accessed from the RHS are called *goals*. They both operate on a data base composed of objects for which attributes and values have been defined. An example of a simple RITA agent is shown in Fig. 1. When this agent is executed, Rule 1 fires, because all three of its premises are true. Firing the rule consists of executing its actions, in this case a *deduce* and a *send*. When the *deduce* is executed, it initiates a deduction for the type of the operating system. This means that all goals are searched to find one whose set of actions modifies the type of the operating system. Since Goal 1 is applicable, it is used to deduce this information, and the data base is automatically updated once the information is obtained. Finally, the second action of Rule 1, the *send*, is executed and the sentence "This is a TENEX system" is printed at the user's terminal.

This report is concerned with the design and development of RITA agents that can acquire knowledge through interactions with users or external computer programs. The knowledge can be represented either as facts stored in a data base or as procedures for data manipulation which are stored in production-rule form. The general problem of knowledge acquisition within the personal computer framework is discussed in Sec. II. The remainder of the report consists of specific examples of agents that can acquire knowledge.

DATA BASE

```
OBJECT operating-system:
        host-computer       IS "PDP-10",
        prompt-character    IS "@",
        net-address         IS "sumex-aim";

OBJECT operating-system:
        type                IS "UNIX",
        host-computer       IS "PDP-11",
        prompt-character    IS "%",
        net-address         IS "rand-unix";
```

RULE SET

```
RULE 1:
    IF: THERE IS an operating-system
        WHOSE host-computer IS KNOWN
        AND WHOSE prompt-character IS KNOWN
        AND WHOSE type IS NOT KNOWN

    THEN: DEDUCE the type OF the operating-system
      AND SEND concat( "This is a ",
            the type OF the operating-system,
            "  system") TO user;

GOAL 1:
    IF: the host-computer of the operating-system IS "PDP-10"
      AND the prompt-character of the operating-system IS "@"

    THEN: SET the type OF the operating-system TO "TENEX";
```

Fig. 1—An example of a RITA agent using both LHS-driven and RHS-driven production rules. (RITA reserved words are shown in upper case without quotation marks.)

II. KNOWLEDGE ACQUISITION

Creating and debugging programs is clearly an important and difficult task, one that consumes an inordinate amount of time and energy. For this reason, much effort has been devoted to developing methods for making programming easier. One thrust of this activity has been the creation of high-level, special-purpose languages such as PLANNER (Hewitt, 1972), CONNIVER (Sussman and McDermott, 1972), QA4 (Rulifson et al., 1972), and KRL (Bobrow and Winograd, 1976), to mention just a few. These languages have sophisticated mechanisms built in for commonly used processes such as pattern-matching and backtracking.

Another approach to the problem has been the development of techniques for automatically creating working computer programs. This research has taken several different paths: program synthesis by selection (Goldberg, 1975), where the user provides information about his particular application and the system integrates that information into an existing program or puts together a program from modules selected by the user; program synthesis from rule-based specifications (Green and Barstow, 1975; Manna and Waldinger, 1975), where rules about programming are accessed by pattern-directed invocation; program synthesis through natural-language understanding (Balzer, 1972,1973; Buchanan, 1974; Lenat, 1975), where the task is specified interactively or in some domain-dependent language and the system must understand the task and map it into executable code; and program synthesis from examples (Biermann and Krishnaswamy, 1974; Biermann, 1976; Siklossy and Sykes, 1975; Green, et al., 1974; Green, 1976), where the partial states of the processing are presented and the system must use these to generate the program.

EXEMPLARY PROGRAMMING

Exemplary programming (EP) is a type of program synthesis by example that relies on program specification from examples or traces of the activity to be performed. The user tells the EP system what he wants done by actually doing it rather than by presenting the system with a general description of the process. The primary difficulty with this approach is in providing the EP system with techniques for making generalizations about what the user has done. This can be accomplished either by building into the system domain-specific knowledge for making inferences about which aspects of the process are invariant and which are not, or by permitting the system to interact with the user to extract this information. A program is usually thought of as procedural knowledge, but the distinction between procedural and declarative knowledge is often blurred. Although we will distinguish between the acquisition of procedural knowledge (program creation) and the acquisition of declarative knowledge (data acquisition), both processes will be termed *knowledge acquisition*, and the acquisition of procedural or declarative knowledge through analysis of examples obtained via man-machine interaction will be called exemplary programming.

In its most basic form, EP may be defined as the user and the machine working together to provide the machine with the knowledge it needs to perform some task

for the user. This report explores the EP approach to providing the machine with information, as well as several more conventional methods, within the framework of the interactive transaction agent, called a *user* agent.

USER AGENTS

A user agent is a program that can act as an interface between the user and the local or external computer systems he might want to access. This type of program is usually small and often resides in a user's terminal (or in a portion of a central timesharing system). The agent typically displays many of the characteristics of a human assistant. For example, it may have the ability to carry on a dialogue with either the user or external computer systems. It may have specific knowledge about particular users, i.e., user A wants his mail retrieved from system X every Friday at 10 A.M., or user B tends to accidentally invert pairs of letters when he types, often typing "lgoin" for "login." The user agent may also have the ability to interact with other agents that are currently operational in the terminal or even to create other agents and initiate their operation.

User agents can be classified along two different dimensions: one which describes the way the agent relates to the user, and another which describes the knowledge acquisition capability of the agent. The agent can relate to the user in a very direct way and hence be considered an active agent, or it can relate in a very indirect way and be considered a passive agent. Neither of these classifications is absolute; actual agents can exhibit both active and passive properties to a varying degree.

The active agent, as shown in Fig. 2, stands between the user and the external system, hiding the characteristics of that system. The agent may carry on a dialogue with the user in one language and with the external system in another, never permitting the user to talk directly to the external system. This type of agent can take the user's input, translate it into something the external system will understand, and immediately send it to the system. Or it may gather a large amount of knowledge from the user, process it, and then use it later to perform some task for the user. Alternatively, the agent may never interact with the user but may instead perform some periodic, routine task and leave the result where the user can find it when he wants it.

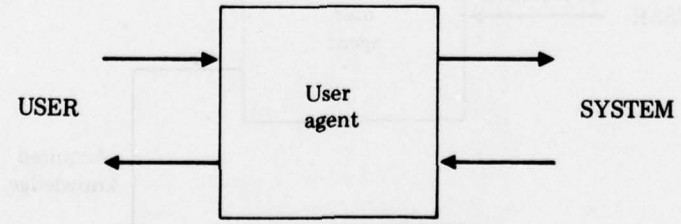


Fig. 2—Active user agent

In contrast, a passive agent, as shown in Fig. 3, passes the user's input directly to the external system and passes the system's reply directly back to the user. Thus it maintains a low profile, giving the user the impression that he is talking directly to the other system. The passive agent tends to let the user take the initiative and guide the course of the interaction, i.e., the user may ask the agent questions, give it commands, and generally maintain control of the situation. The active agent, on the other hand, may try to accomplish a particular task in some prespecified way and often maintains control, by querying the user when task-specific information is needed and then supplying him with the result when the task is completed.

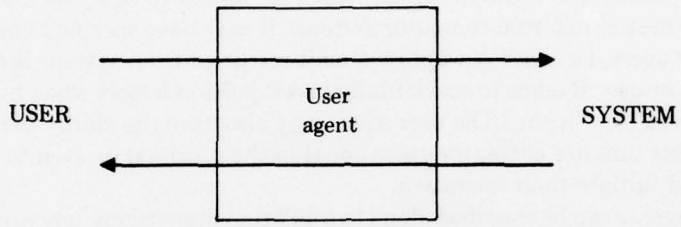


Fig. 3—Passive user agent

If the user agent is classified in terms of its knowledge acquisition capability, it may be termed either static (incapable of permanently acquiring new knowledge) or dynamic (able to acquire new knowledge and later use it in new situations). In this report, we are particularly concerned with dynamic agents and will further classify them as being either creative or adaptive. A creative agent is one that in some sense creates, builds, or modifies *another* agent. It acquires information and then represents this information in a form that can be used by some other agent. Figure 4 illustrates the operation of a creative agent.

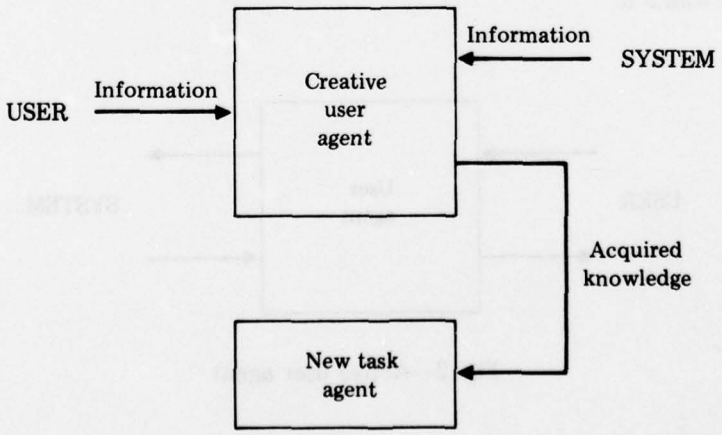


Fig. 4—Operation of a creative user agent

The creative agent contains rules that tell it how to map the information it receives from the user and from external systems into knowledge that can be used by the new agent. This knowledge is either in declarative form (data) or procedural form (rules) and is saved as part of the new agent.

The adaptive agent is similar to the creative one in that it also maps external information into permanent data or rules. The difference is that the adaptive agent modifies *itself* rather than another agent, as shown in Fig. 5. Dynamic agents can also be classified as acquiring either data or both program and data. Thus a creative agent that mapped its knowledge into both rules and data objects for a new RITA agent would be classified as a creative/program type. Conversely, an adaptive agent that mapped its knowledge into new data for itself but not into new rules would be classified as an adaptive/data type of agent.

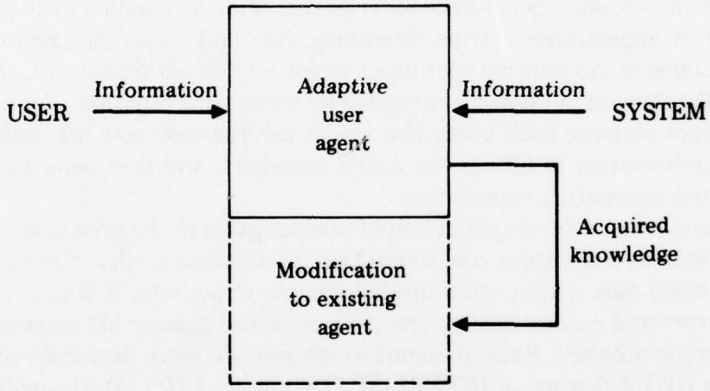


Fig. 5—Operation of an adaptive user agent

APPLICATIONS

User agents can perform a variety of tasks. One important task is that of interfacing the user to external systems, i.e., providing the user with active help in learning and using complex remote systems. A good example of this type of remote system is the New York Times Information Bank (NYTIB), which contains abstracts of recent news articles. In this situation, the agent could query the user about what kinds of abstracts he wanted to retrieve and could then access the NYTIB and perform the retrieval. Alternatively, it could act as an interface between the user and the NYTIB, answering all the queries from the NYTIB that it could and mapping those it could not answer into a form intelligible to the user so that he could answer them in real time. In general, when the user is interacting with an external system whose characteristics he wants to learn, such as an interactive programming-language interpreter, he should have available a passive help and tutoring facility which does not mask any of the characteristics of the system. If he wants only to expedite the interaction, then an active agent is usually appropriate.

Another important type of task that user agents can perform is message-handling (Anderson, 1977). This includes reading and analyzing incoming mail and

informing the user when important or high-priority messages arrive. Agents could also be designed to initiate mail under certain circumstances and even to respond to the replies that were elicited. For example, if the agent were trying to use a local system, such as a network control program, and got error messages when attempting to call it, the agent could send a message to the person responsible for maintaining the system, asking for help or maintenance. Sophisticated agents could be used to help prepare outgoing mail; they could assemble and update form letters or even query the user to obtain information needed to create reactive messages (see Sec. IV).

User agents are particularly useful for providing mundane, periodic services such as network accessing, secretarial chores, and accounting operations. Typical network-accessing operations include sending and retrieving files to and from remote installations, initiating and running jobs such as statistical analysis programs on a remote computer, and managing transactions between services distributed over the network. Secretarial services that could be handled include reminding the user of appointments, trips, deadlines, etc., and recording the outcomes for future reference. Accounting operations include filling out timesheets, travel forms, or any other type of questionnaire needed for accounting purposes. For example, the agent might activate itself every two weeks, ask the user how his time was spent, use the information to fill out the user's timesheet, and then send the completed form to the accounting department.

A final, more esoteric application of user agents is in the area of error detection and correction. If an agent could maintain an accurate model of the user and his current intentions within some limited domain of activity, it could recognize the user's errors and automatically correct them before passing his response on to the system being accessed. Such an agent would provide more personalized correction than the DWIM feature in INTERLISP (Teitelman, 1975), which applies general-purpose spelling and syntax correction routines to input dealing with defining and debugging of LISP programs.

RITA agents are like any other computer programs in the sense that once they are designed to perform some specific task, they must be modified or reprogrammed before they can perform different or additional tasks. However, the language has been designed to make this modification or reprogramming as simple as possible, mainly through the provision of a modular, stylized syntax which can be written to resemble a few basic English constructs. Still, the job of making major modifications to a RITA agent or of creating a new RITA agent to perform some new task is difficult for the novice. The solution proposed here is to handle the task of changing or creating a RITA agent in exactly the same way the tasks mentioned above are handled: by using a RITA agent to assist the user. Creative and adaptive agents—relatively sophisticated knowledge-acquiring agents—can be used to perform this type of complex information processing.

To illustrate the feasibility of this approach, we have devoted the remainder of this report to the discussion of four dynamic RITA agents which have been implemented and their offspring, static RITA agents that perform simple tasks for the user. These agents are summarized in Table 1.

EP-1 watches a user perform a series of operations on the computer and then attempts to write a program (another agent called TASK) to perform the same task. TRANSFER, the self-modifying agent, performs tasks with the help of the user, such

Table 1

RITA AGENTS

Name	Task	Relation to User	Knowledge Acquisition	Offspring
EP-1	Create an agent to perform same job for the user	Passive/active	Creative/program	TASK
TUTOR	Modify an agent so it will be capable of tutoring the user	Passive	Creative/data	TEACH
TRANSFER	Assist the user in retrieving files via the net	Active	Adaptive/program-data	TRANSFER
WRITER	Assist the user in creating a reactive message	Active	Creative/data	MESSAGE
TASK	Perform some low-level job for the user	Active	Static	None
TEACH	Assist the user in learning some external system	Passive	Static	None
MESSAGE	Interact with the recipient of a message and send the result to the user (sender)	Active	Static	None

as a file transfer from one site to another, and then modifies itself so that the next time it is asked to perform the task it can do so with less help from the user. TUTOR watches an expert demonstrate how to use an external system and then modifies TEACH so it can perform the demonstrations itself. TEACH is a teaching agent that helps naive users become familiar with interactive computer languages or local operating systems. WRITER is the agent that assists the user in creating MESSAGE, the reactive-message agent. MESSAGE carries on a dialogue with the recipient of the message, automatically sending the information elicited from the recipient back to the sender of the message.

The first four agents in Table 1 are dynamic, that is, capable of producing or modifying offspring (other agents). The last three agents are the offspring produced by the dynamic agents. Section III describes the EP-1, TASK, and TRANSFER agents; Sec. IV, the TUTOR and TEACH agents; and Sec. V, the WRITER and MESSAGE agents. Concluding remarks are presented in Sec. VI.

III. PROGRAM CREATION THROUGH USER AGENTS

Program creation within the production-system framework can proceed quite naturally in discrete steps or increments. This incremental programming can be delimited by feedback to the system that describes the problems inherent in the current code (Waterman, 1975, 1976a). The system would then in effect be writing and debugging the program simultaneously. The two RITA agents described in this section use a slightly different approach. These agents create the program incrementally, but debugging (in the form of extending and refining the program) is done as a later step. The rationale is that it is easier and more efficient to create a complete program that performs the task correctly most of the time, then later apply sophisticated debugging techniques to fine-tune the program so that it will produce the correct output for its range of likely inputs. Here, our concern is with conceptual errors rather than syntax errors, since the program-creating agents can be designed to avoid making syntax errors.

The first two agents discussed below, EP-1 and TRANSFER, are program creators, i.e., they transform procedural knowledge into RITA rules or goals. The organization and operation of these two user agents will be described in detail.

EP-1: AN EXEMPLARY PROGRAMMING AGENT

EP-1 is a RITA agent that watches the user interact with an external computer system and then creates a new agent to perform the task it saw the user perform. The information used to create the new TASK agent is derived two ways: (1) passively, as EP-1 observes the actions of the user and the associated responses of the external system, and (2) actively, as EP-1 queries the user about what he is doing during the interaction. Thus EP-1 has both passive and active characteristics, although it is primarily a passive agent. The information flow between the user, EP-1, and the external system is diagrammed in Fig. 6.

Initially, the user sends task commands to the external system and receives system responses, using EP-1 as an intermediary that passes the messages back and forth and asks questions about what is happening. The knowledge acquired from this interaction is used to create the TASK agent incrementally, as the user performs the task. When the user is finished, TASK is complete and can then be used to perform the task, as shown by the dotted lines in Fig. 6. TASK may itself be an interactive program that queries the user when it needs help or information.

Modes of Operation

EP-1 currently has one mode of operation, the acquisition mode, but will eventually incorporate a training mode. In the acquisition mode, EP-1 watches the user perform some interactive task, asks the user appropriate questions about the task, and then writes a set of RITA rules to perform that task. In the training mode, the situation is reversed: The user watches EP-1 run a rule set which performs some interactive task. Whenever the rule set responds incorrectly to the system it is

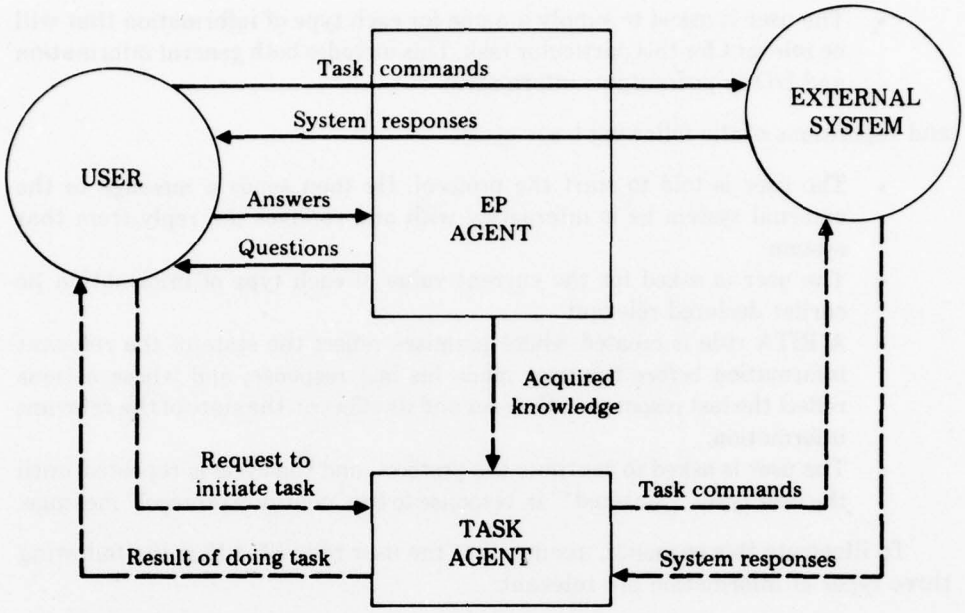


Fig. 6—Information flow in EP-1 agent

interacting with, the user tells EP-1 what the correct response should have been (and possibly why it should have been made), and the EP agent modifies the rule set so that it will thereafter make the correct response in that situation. The acquisition mode is essentially a program-writing mode in which EP-1, with the help of the user, is able to write a program from scratch. By contrast, the training mode is a debugging mode, in which EP-1 can debug the program it creates during acquisition.

Acquisition Mode

Ideally, we would like EP-1 to monitor the behavior of the user, analyze that behavior, and from it create new RITA agents to duplicate the behavior, without having to interrogate the user. The questioning of the user by EP-1 is necessary because the current version is quite general: No domain-specific knowledge is built into the agent. The questioning can be reduced or eliminated by adding such knowledge, but care must be taken to ensure that the rules produced use object, attribute, and value names that are mnemonic, and that the rules have no unnecessary premises.

Current Implementation. As a first pass at developing an EP agent, we have created a working RITA program which operates in the acquisition mode. It watches the user perform a task and then creates a new RITA agent capable of performing the same task. The operation of EP-1 during acquisition consists of an initialization step:

- The user is asked to supply a name for each type of information that will be relevant for this particular task. This includes both general information and I/O (input/output) information.

and repetitions of the following basic cycle:

- The user is told to start the protocol. He then sends a message to the external system he is interacting with and receives the reply from that system.
- The user is asked for the current value of each type of information he earlier declared relevant.
- A RITA rule is created whose premises reflect the state of the relevant information before the user made his last response, and whose actions reflect the last response of the user and its effect on the state of the relevant information.
- The user is asked to continue the protocol, and the cycle is repeated until the user types "*finished*" in response to the "continue protocol" message.

To illustrate this sequence, assume that the user tells EP-1 that the following three types of information are relevant:

- The name of the current system *(the name of the operating system or program the user or agent is currently accessing.)*
- The state of the agent *(a term describing what the agent is currently trying to accomplish, i.e., "logging in," or "giving password.")*
- The value of the response *(the response the user or agent receives from the external system, i.e., "host:" from the FTP program.)*

Furthermore, the user specifies that "value of response" is I/O information. After EP-1 tells the user to start the protocol, assume that he types a carriage return, gets back a "%," and tells EP-1 that the name of the current system is "unix" and the state of the agent is "use the ftp program." In this situation, the dialogue shown in Fig. 7 would lead to the creation of a new rule.

```
    EP-1: Please continue the protocol
   USER: ftp
 SYSTEM: HOST:
    EP-1: What is the name of the current-system?
   USER: file transfer program
    EP-1: What is the state of the agent?
   USER: give the host name
```

Fig. 7—Dialogue needed for rule creation

At this point, EP-1 creates a rule using the information just elicited from the user and the information elicited in the previous cycle. The rule created would be similar to the one below:

RULE n

IF: the name of the current-system is "unix"
and the state of the agent is "use ftp"
and the value of the response contains a {"% "}

THEN: set the name of the current-system to "file transfer program"
and set the state of the agent to "give the host name"
and set the value of the response to " "
and set the reply of the agent to "ftp";

A rule at the beginning of the rule set being created sends the reply of the agent, in this case "ftp," to the external system. Thus when rule n is fired, it will send "ftp" to unix and set the values of the relevant objects appropriately.

As the user is interacting with the external system, the EP agent is monitoring the interaction, asking questions, creating RITA rules, and storing them in a disk file. When the user is finished with his task, the EP agent is also finished with the job of creating the new rule set or agent, and this rule set is available for immediate use. The user is not required to examine or modify the rule set in any way before he uses it to accomplish the desired task.

To illustrate more fully the operation of EP-1 during the acquisition phase, an actual protocol of a very simple user-learner interaction is shown below. The task is to determine who is currently using the unix system. User input is shown in italics, EP agent response in normal type, and system response in boldface.

**run;*
What task are you going to perform?
Seeing who is using unix
What is relevant?
the name of the current system
What else?
the state of the agent
What else?
nothing
What do you want to call the I/O?
the value of the response

Please start the protocol to be learned

(carriage return)
%
What is the name of the current system ?
unix

What is the state of the agent ?
use the who command

Please continue the protocol to be learned

who
k2240 ttya Jul 6 15:26
don ttyf Jul 6 15:22
%
What is the name of the current system ?
(carriage return)

What is the state of the agent ?
quit

Please continue the protocol to be learned

finished
done

The following RITA rules were written by EP-1 during the above interaction:

```
[OBJECTS:]

OBJECT agent<1>: ;

OBJECT current-system<1>: ;

OBJECT response<1>:
        value   IS " ",
        input   IS " ",
        count   IS "0";

[RULES:]

  RULE 1:
       IF: the prompt OF the agent IS KNOWN
     THEN: SEND the prompt OF the agent TO user
           & RECEIVE the NEXT {ANYTHING 'line-contents'
             FOLLOWED BY "^j"}† FOR 15 SECONDS FROM user
           & SET the the new-reply OF the agent TO 'line-contents'
           & SET the prompt OF the agent TO NOT KNOWN;

  RULE 2:
       IF: the new-reply OF the agent IS KNOWN
           & the new-reply OF the agent IS NOT " "
     THEN: SET the reply OF the agent TO the new-reply OF the agent
           & SET the new-reply OF the agent TO " ";

  RULE 3:
       IF: the reply OF the agent IS KNOWN
     THEN: SEND the reply OF the agent TO system
           & SEND " " TO user
```

†This statement reads the next line typed in by the user and calls it 'line-contents', stripping off the terminating carriage return (represented by " ^ j").

```
        & SEND concat( "sent: ", the reply OF the agent ) TO user
        & SET the reply OF the agent TO NOT KNOWN;

RULE 4:
    IF: the name OF the current-system IS NOT KNOWN
        & the state OF the agent IS NOT KNOWN
        & the value OF the response CONTAINS {" "}
  THEN: SET the name OF the current-system TO "unix"
        & SET the state OF the agent TO "use the who command"
        & SET the value OF the response TO " "
        & SET the reply OF the agent TO " ";

RULE 5:
    IF: the name OF the current-system IS "unix"
        & the state OF the agent IS "use the who command"
        & the value OF the response CONTAINS {"% "}
  THEN: SET the state OF the agent TO "quit"
        & SET the value OF the response TO " "
        & SET the reply OF the agent TO "who";

RULE 6:
    IF: the state OF the agent IS "quit"
        & the value OF the response CONTAINS {"% "}
  THEN: SEND "Seeing who is using unix has been completed"
          TO user
        & RETURN SUCCESS;

RULE 7:
    IF: the input OF the response IS " "
        & the count OF the response IS LESS THAN 20
  THEN: RECEIVE FOR 5 SECONDS FROM system
          AS the input OF the response
        & SET the count OF the response TO
          1 + the count OF the response;

RULE 8:
    IF: the count OF the response IS LESS THAN 20
  THEN: SET the value OF the response TO
          concat( the value OF the response,
            the input OF the response )
        & SET the input OF the response TO " "
        & SET the count OF the response TO 0
        & SEND concat( "got: ", the value OF the response )
          TO user;
```

The first three rules are standard rules always generated to handle the mechanics of sending messages specified by the other rules to either the user or the external system. The last two rules are used to look for the response from the external system and to limit the time expended so that the agent won't go into a loop if the external system never bothers to respond. Rules 4, 5, and 6 are the ones that define the actual

task: that of checking to see if unix is running (entering a carriage return to see if it elicits a %), using the "who" command, and then halting.

The actual protocol produced by running the above rule set created by EP-1 is shown below. User input is shown in italics, EP agent response in normal type, and system response in boldface.

```
* run;

sent: (carriage-return)
got: %

sent: who
got: jal        tty9 Jul 18 18:35
     leone      ttyd Jul 18 17:44
     lai        ttyi Jul 18 17:30
     don        ttym Jul 18 16:38
%
Seeing who is using unix has been completed

* exit;
```

Appendix A presents a complex RITA agent written by EP-1. This task agent automatically retrieves files from remote sites on the ARPAnet and copies them into a RAND unix file. It interrogates the user, asking him for the following information:

- Remote-site name.
- Account name at that site.
- Password at that site.
- Name of file to be retrieved.
- New name for retrieved file.

If the user fails to respond to a question within a reasonable length of time (currently 15 seconds), the agent goes on to the next question and assigns the default answer for the unanswered question. As the agent obtains the information it needs, it calls the local file-transfer program, answers its questions, and initiates the retrieval. When the retrieval is complete, the agent informs the user of this fact and halts.

The file-transfer agent shown in App. A was written by EP-1 as it watched the user transfer a file from the Carnegie-Mellon University PDP-10 system to the RAND unix system. During the interaction, the user indicated which items of information were to be considered variables by appending a colon and prompt to the user responses that defined those items. Figure 8, which is a continuation of the dialogue of Fig. 7, illustrates this process.

```
   EP-1: Please continue the protocol
   USER: cmu-10a:What system shall I retrieve the file from?...
 SYSTEM: Connections established.
         300 CMU10A 7.U5/DEC 5.06B FTP Server 4(30)
         >
   EP-1: What is the name of the current-system?
   USER: (carriage-return)
   EP-1: What is the state of the agent?
   USER: login
```

Fig. 8—Dialogue illustrating creation of rules with prompts

The rule that would be created from the dialogue of Fig. 8 is shown below (see App. A, Rule 5):

```
RULE m
     IF: the name of the current-system is
           "file transfer program"
         and the state of the agent is "give the host name"
         and the value of the response contains {"Host: "}

   THEN: set the state of the agent to "login"
         and set the reply of the agent to "cmu-10a"
         and set the prompt of the agent to
           "What system shall I retrieve the file from?...";
```

A rule at the beginning of the agent being created sends the value associated with "the prompt of the agent" (in the above example, the "What system ..." message) to the user and then passes the user's reply on to the external system. If the user does not respond to the prompt in time (or responds with a carriage return), the agent replies for the user, sending the value associated with "the reply of the agent" (in the above example, "cmu-10a") to the external system. The above rule does not reset "the name of the current-system," because the user indicated that this information was not currently relevant by entering a carriage return rather than the actual name of the current system during the dialogue that led to the creation of the rule (see Fig. 8).

The operation of the task agent created by EP-1 is illustrated in the protocol given in App. B. Note that in some cases the messages sent by the agent were garbled during transmission, and the agent received unexpected error messages from the external system. In these cases, the agent correctly backed up to the appropriate spot in the sequence of operations and repeated the messages until they were correctly transmitted and received.

Expansion. This initial version of an EP agent could be expanded in a number of ways to make the acquisition phase more effective. First, a more sophisticated method could be developed for determining what part of the external response (the I/O information) should be used in the rule being created at each step. In the previous example there was no problem: The response was "% ," so the entire response could be used in the left-hand side of the rule as part of the premise.

However, when the response consists of many lines and is not always exactly the same for the same input to the external system, then something more is needed. The current implementation of the EP agent handles the problem by simply using the first *n* characters of the last line the external system gives as a response. This could be slightly improved by using the first *n* characters of both the first and last lines. However, to effect any significant improvement it would be necessary to build into EP-1 specific knowledge about the tasks it could be asked to learn, plus rules telling it how to decide what components of the system response are likely to be invariant (or static information from several different examples would have to be induced). The alternative would be to have the EP agent learn which components are invariant through empirical verification, which might prove time-consuming.

Second, the EP agent could be modified so that it would not ask the user for the values of the relevant objects at each step, but instead would supply these values itself. This would reduce the workload on the user but would introduce some difficulties: (1) The mnemonics created by the agent would not be as intelligible as those created by the user, unless the agent had clever heuristics for inferring good mnemonics from the trace of the user-system interaction, and (2) global knowledge about the task being performed would have to be built into the EP agent so that it could infer the values of the relevant objects, e.g., it would have to know that if the user types a *carriage return* and gets back a "% ," the name of the current system is *unix*. This type of inference is currently made by TRANSFER (the agent discussed next in this section).

Finally, the EP agent could be made not only creative but also adaptive. The result would be an agent capable of creating new agents, debugging them, and, in the course of the debugging, modifying itself so that it would subsequently generate agents with fewer bugs.

Training Mode

The training mode for the EP agent is not yet implemented, but it will feature interactive rule acquisition and modification similar to that in TEIRESIAS (Davis, 1976). What follows is a partial specification of the features needed in this mode. In the training mode, the agent, with the help of the user, will debug a given RITA rule set by running the rule set and making corrections to it whenever it makes an error. Each time the rule set makes a decision (fires a rule), the EP agent will give the user an opportunity to state that an incorrect decision was made and inform the agent of the correct decision. The EP agent will then correct the rule set being debugged by either adding a new rule or modifying an existing one. A hypothetical protocol for this process is shown below:

SYSTEM: %
AGENT: sent: ftp
EP: was this ok?
USER: yes

SYSTEM: Host:
AGENT: sent: cmu
EP: was this ok?
USER: cmu-10b not cmu

Now the EP agent modifies the new agent by changing all occurrences of *cmu* in the rule that just fired to *cmu-10b*; it then starts the agent running again. If instead the user says, "ask for site name," the EP agent adds to the current rule an action that queries the user for the site name instead of considering it invariant:

AGENT: sent: cmu-10b
EP: was this ok?
USER: yes

SYSTEM: *Connection established.*
300 CMU10B 7.U5/DEC 5.06B FTP Server
>
AGENT: sent: retrieve news ftpdata
EP: was this ok?
USER: do: user A330DW28

Now the EP agent adds a new rule that says, in effect, when you get a "Connection established" message send "user A330DW28" to the remote system. It then starts the agent running again.

AGENT: sent: user A330DW28
EP: was this ok?
USER: yes

During training, the EP agent will be able to correct errors that were made during the acquisition phase. As illustrated above, two types of errors will be corrected: those caused by the user giving the EP agent incorrect information during acquisition (e.g., cmu instead of cmu-10b) and those caused by the user failing to supply the EP agent with information about what to do in a given situation (e.g., login before you start retrieval). The EP agent will also be able to create new rules that are modifications of existing rules. For example, if the above protocol continued with the agent sending "abc" for the password on cmu-10b and the current host was not cmu-10b, the user could indicate that the password was wrong in this instance with "bcd this time, not abc." The EP agent would then query the user to find that the current host (say, SRI-AI) was relevant at this point and add a rule saying, in effect, that if the host is SRI-AI, then the password is "bcd."

TRANSFER: A FILE-TRANSFER AGENT

TRANSFER is a sophisticated user agent designed to help the user transfer files across the ARPAnet. It differs from the file-transfer agent created by EP-1 in three important respects: First, it contains information about the characteristics of various operating systems, which enables it to recognize which system it is interacting with and take the appropriate action. This is particularly important when system errors occur and the agent ends up talking to an unexpected system. It can recover from such an error only if it knows that an error has occurred and recognizes the system with which it is currently interacting.

The second way in which TRANSFER differs from the EP-1-created file-transfer agent is that it has a goal-driven component. TRANSFER initially attempts to

deduce the information it needs for file transfer—the file name, site name, account name, etc. If it fails to find the information in the data base and cannot deduce it, TRANSFER automatically queries the user for the information.

The third and most important difference between the two file-transfer agents is that TRANSFER is adaptive, i.e., it can modify itself on the basis of knowledge gained through interaction with the user. Once it obtains the information it needs to login to a particular site, it retains it in the form of data objects and goals so it won't have to query the user again at some later date for the same information. To illustrate, a protocol of an actual user-agent interaction is shown below. User input is shown in italics, TRANSFER response in normal type, and system response in boldface:

% rita use.ft.new

Ready...

** run;*
What is the name of the current-file?
Response: *mail*
What is the site-name of the current-file?
Response: *aim*
What is the site of the current-file?
Response: *sumex-aim*
What is the account-name of the current-file?
Response: *aihandbook*
What is the password of the current-file?
Response: *abcdef*
What is the directory of the current-file?
Response: *<aihandbook>*
Will now attempt to retrieve file <aihandbook>mail
from the sumex-aim system

Sent: ftp

Received from net: Host:

Sent: sumex-aim

Received from ftp: Connections established.

300 SUMEX-AIM FTP Server 1.44.0.0 – at MON 19-JUL-76 11:37-PDT
>

Sent: user aihandbook

Received from ftp: 330 User name accepted. Password, please.
>

Sent: pass abcdef

Received from ftp: 230 Login completed.
>

Sent: retrieve <aihandbook>mail ftpdata
File has been transferred

Received from ftp: 255 SOCK 3276965379
250 ASCII retrieve of <AIHANDBOOK>PEOPLE.;1 started.
252 Transfer completed.
>

Sent: bye
Job finished
done

* *exit;*
exiting.

During the course of the above interaction, TRANSFER permanently acquired the data objects and goals shown in Fig. 9. This knowledge enabled the agent to deduce the information it needed the second time the user asked for the file to be transferred, as shown by the partial protocol below:

% rita use.ft.new

Ready...

* *run;*
What is the name of the current-file?
Response:*mail*
Will now attempt to retrieve file <aihandbook>mail
from the sumex-aim system

Sent: ftp

From this point on, the protcol is the same as the one shown earlier for file retrieval using TRANSFER.

The knowledge mapped into RITA goals by TRANSFER could have been represented more simply as RITA data objects. However, the use of goals provided the agent with the implicit question-asking capability of the goal-driven RITA monitor. Thus no rules or goals were needed that explicitly directed the agent to ask the user for information. This request was handled by simply telling the agent to deduce the information when it was needed.

```
OBJECT known-files<1>:
        names.  IS ( "mail");

OBJECT known-sites<1>:
        names.  IS ( "sumex-aim");

OBJECT sumex-aim<1>:
        aliases IS ( "aim", "sumex-aim" ),
        files   IS ( "mail" );

  GOAL 5:
        IF:  the name OF the current-file IS IN
               the files OF the sumex-aim
      THEN:  SET the site OF the current-file TO "sumex-aim"
             & SET the site-name OF the current-file TO "sumex-aim";

  GOAL 6:
        IF:  the site OF the current-file IS "sumex-aim"
      THEN:  SET the password OF the current-file TO "abcdef"
             & SET the account-name OF the current-file TO
             "aihandbook"
             & SET the account-preface OF the current-file TO "user"
             & SET the password-preface OF the current-file TO "pass";

  GOAL 7:
        IF:  the site-name OF the current-file IS IN
               the aliases OF the sumex-aim
      THEN:  SET the site OF the current-file TO "sumex-aim";

  GOAL 8:
        IF:  the name OF the current-file IS "mail"
      THEN:  SET the directory OF the current-file TO "<aihandbook>";
```

Fig. 9—Data and goals learned by TRANSFER agent

IV. DATA ACQUISITION THROUGH USER AGENTS

Data acquisition is a much easier task than program creation—it consists simply of adding new knowledge to the existing data base. However, user agents that employ data acquisition exhibit many of the dynamic characteristics of the more powerful program-creating programs. The trick is to organize the agent being created so that the fixed, unchanging portion of the knowledge it contains is procedural, that is, represented as rules or goals. The fluctuating, context-dependent portion is then represented as data for the procedures to operate on. This was the approach taken in the design of both TEACH and MESSAGE. TUTOR, the agent that creates TEACH, generates the data base for TEACH, but not the rules that TEACH uses. Similarly, WRITER, the agent that creates MESSAGE, generates only the data base for MESSAGE. This results in the creation of useful programs because the data bases of TEACH and MESSAGE contain all the context-dependent knowledge about the task to be performed.

TEACH: A TUTORING AGENT

TEACH is a RITA agent that can help a user learn how to use interactive computer languages or local operating systems. It works by acting as an invisible interface between the user and the system the user is trying to learn, passing all standard user-originated messages to the system and all system-originated messages back to the user. In addition, TEACH recognizes special user-originated messages and responds to them either by sending appropriate text to the user or by conducting interactive demonstrations of the system's capabilities. Thus it appears to the user that the system he is trying to learn is able to explain and demonstrate its own operation.

The TEACH agent consists of 14 short RITA rules which are somewhat domain-independent, that is, they can be used to teach a variety of languages or operating systems. The syntax of the special messages TEACH recognizes is quite simple: either "show me <arbitrary string of char's>" to elicit an interactive demonstration, "again" to elicit a new demonstration of whatever was last demonstrated, or "exit"* to terminate the TEACH program. The TEACH data base consists of RITA objects which are domain-dependent, and thus a different data base must be supplied for each new language or operating system taught by TEACH.

The TEACH data base contains three types of objects: *intros*, *texts*, and *demos*. Each intro contains a piece of text that is elicited when the user types the name of the intro. A typical name is HELP, which should supply the user with a message explaining what special messages TEACH recognizes. Each *text* object also contains a piece of text, but this is elicited only when the user types "show me <name of text>." For example, if one *text* object is named "function names," then when the user types "show me function names" he should elicit a list of all pertinent function

*If the language being taught contains these messages as valid commands, they can be modified by adding special control characters.

names. Each demo object contains a list of one or more sequences of commands. When the user types "show me <name of demo>," TEACH sends one of the sequences of commands to the system in such a way that they appear to be user-originated messages. The system-originated replies are returned to the user, and thus the user sees a live demonstration of the system's capabilities.

When the user loads TEACH, the intro object with the name "what?" is automatically accessed and its text displayed. Thus a user-TEACH interaction might proceed as follows (user responses are shown in italics):

rita use.teach	(user loads the TEACH agent)
Type "help" for help	(automatic intro message from agent)
help	(user types "help")
Type "show me" followed by "functions" or (any function name).	(agent displays text of the INTRO named "help")
Show me functions	(user types "show me functions")
The available functions are plus, minus, and div	(agent displays example of TEXT named "functions")
show me plus	(user types "show me plus")
[demo of plus]	(agent sends commands and receives responses from system, displaying all I/O to user)

TEACH uses a very simple control mechanism for its demonstrations: It cycles through a list of commands, sending one to the system, receiving the response from the system, and then sending the next command in the list, regardless of the value of the response. This works well for teaching a programming language or local operating system because the number of possible responses for any given command is low (usually 1). However, this technique could not be used to teach a user how to interact with a complex system, such as the ARPAnet, which has many possible responses for any given command. Teaching this type of system requires an agent that sends commands based on the responses it receives, as does the TRANSFER agent.

There are a number of advantages to using TEACH in conjunction with the typical printed reference manual. First, the information is on-line, so the user always has it available when he is using the system. It occurs within the real on-line context, so the user can freely mix experiment and query. Second, the demonstrations are live, not canned, which means that if the system changes, the changes will be seen the next time the demonstration is run. A reference manual with a listing of a demonstration would simply become out-of-date. Finally, the demonstration is a nice way to show the user how to make use of the facilities described in the

reference manual. Just giving the user a definition of a command or function usually does not show him how to apply it and make proper use of it in conjunction with the other system facilities.

A protocol of an actual user-TEACH interaction is shown in App. C. This protocol shows how TEACH can be used to teach a novice how to use commands and actions in the RITA system.

TUTOR: A PASSIVE EXEMPLARY PROGRAMMING AGENT

TEACH is itself created by another RITA agent called TUTOR. A primary reason for organizing TEACH as a production system was to facilitate its creation by TUTOR. TUTOR is a passive agent that watches the user demonstrate how to use certain features of a system and then places the information needed to recreate that demonstration into the data base of TEACH. This is another instance of EP. The difference between this application and the use of EP-1 to create agents from examples of programming tasks is that with TUTOR the end result is declarative knowledge in the form of RITA objects, while in the case of EP-1 the end result was procedural knowledge in the form of rules.

A protocol of a user interacting with TUTOR to create a TEACH agent for LISP (McCarthy et al., 1965) is shown in App. D. The use of the TEACH agent created for LISP is shown in App. E. Note that by using TUTOR, a LISP expert who doesn't know RITA can create a RITA agent to teach LISP.

REACTIVE-MESSAGE AGENTS

The reactive message is an offshoot of an earlier concept, the interactive letter (Anderson and Gillogly, 1976a; Standish, 1974), which is a letter organized as a computer program. The recipient of the letter "reads" it by interacting with the program. When the dialogue between the user (recipient) and the letter (program) is concluded, the letter transmits a record of the interaction back to the user who sent the letter.

The reactive message is a particular type of interactive letter, one in which the originator-recipient link is a one-to-many mapping. That is, the message is organized to be general enough to be sent by one originator to many different recipients. It is this one-to-many mapping that makes the reactive message cost-effective. Reactive messages—form letters, questionnaires, timesheets, etc.—have been implemented in RITA as RITA agents which contain not only the message to be transmitted but all the machinery needed to initiate a dialogue with a user and transmit the result back to the sender.

The reactive message has a number of advantages over other, more conventional forms of communication. First, the act of reading the message can automatically generate a reply which is transmitted back to the originator or sender of the message. This means that the sender gets instant response. Also, the recipient doesn't have to worry about organizing and forwarding a reply to the message. Second, a long message with lots of text can be "read" quickly by the recipient, since he doesn't have to look at the entire message. Instead, he takes one path through the message,

reading only text that is relevant to his particular situation or interests. Finally, the reactive message maps the recipient's replies onto a small set of expected responses, and these mapped replies can easily be read and understood by another program or agent. This second agent could process this information (e.g., tabulate the results of all replies to a questionnaire) and transmit the end product to the sender.

There are also certain inherent disadvantages in the use of reactive messages. From the point of view of the sender, the message is difficult to organize and create. Since it is a program, some thought has to be given to the flow of control and how it determines which text and questions will be presented in any given context of recipient replies. Once the message is organized, the physical act of writing the program is still a problem. This can be alleviated somewhat by having a message-writing agent help the sender construct the message, but the work involved in interacting with this agent is still greater than the work involved in creating an ordinary piece of computer mail. Moreover, the reactive message presents some special problems for the recipient. For example, he might like to read the message without responding, or just skim it to decide whether or not he wants to actually read it. An even greater problem is created if he wants to back up to some previous answer and start over. Providing such capabilities would result in a more complex control structure for the reactive-message agent, but it would no doubt be worthwhile in terms of usefulness for the recipient.

Implementation of Reactive-Message Agents

An example of a reactive-message agent is MESSAGE, a RITA agent composed of seven short production rules and a large data base organized as a simple semantic net. These context-independent rules are used to search the data base for the next piece of text, send that text to the user, and then instigate a new search based on the user's reply. All the text of the message is stored in the data base at the nodes of the net, thus each reactive-message agent has a totally different data base in terms of structure and content but exactly the same set of production rules to perform the search.*

A sample reactive-message data base is illustrated in Fig. 10. Note that there are two types of nodes in the net: statement nodes and question nodes. Similarly, there are two types of links: associative and alternative. The associative link connects a statement to a lower node and has no user reply assigned to it. In contrast, the alternative link connects a question to a lower node and always has a user reply assigned to it.

One possible user-agent interaction that this message might evoke is shown below:

AGENT: Dear Professor Smith:

I'm currently trying to organize a
weekly seminar on message-handling
systems. The only available day of
the week is Tuesday.

*The ASKUSER facility in INTERLISP (Teitelman, 1975) also provides a way of defining a user-system dialogue, but it is used more as a sophisticated prompting mechanism than as a message-handling system.

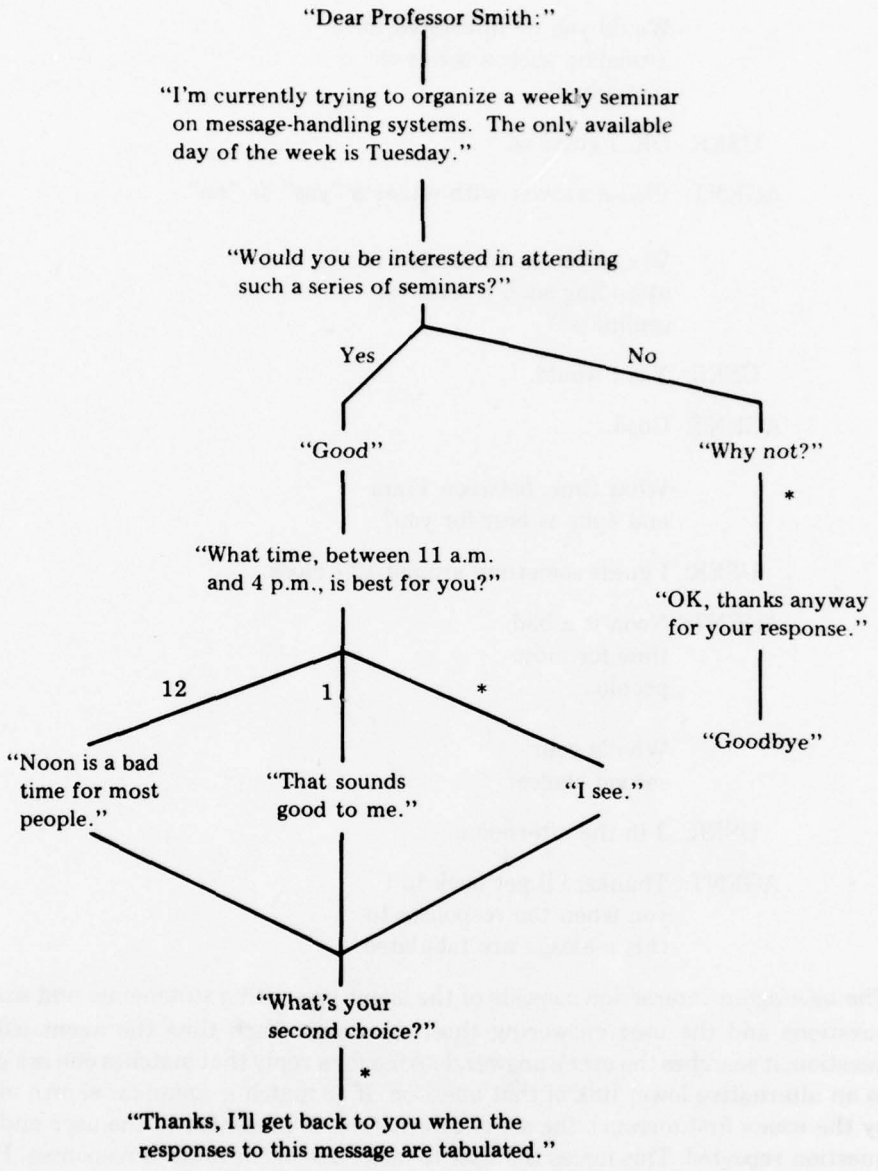


Fig. 10—Data base for a reactive message

Would you be interested in attending such a series of seminars?"

USER: OK, I guess so.

AGENT: Please answer with either a "yes" or "no".

Would you be interested in attending such a series of seminars?"

USER: Yes I would.

AGENT: Good.

What time, between 11am and 4pm, is best for you?

USER: I guess sometime around 12 o'clock.

AGENT: Noon is a bad time for most people.

What's your second choice?

USER: 3 in the afternoon.

AGENT: Thanks, I'll get back to you when the responses to this message are tabulated.

The user-agent interaction consists of the agent presenting statements and asking questions and the user answering those questions. Each time the agent asks a question, it searches the user's answer, looking for a reply that matches one assigned to an alternative lower link of that question. If no match is found (as shown above by the user's first answer), the set of alternatives is presented to the user and the question repeated. This forces the user to make one of the desired responses. However, since it is virtually impossible to predict all reasonable responses the user might want to make, a special category, designated by the asterisk(*), is permitted. This is the catch-all category and matches any answer the user gives. Thus in the above example, if the user had replied "2 o'clock" to the question, "What time, between 11am and 4pm is best for you?," the agent, after finding that neither "12" nor "1" was in the user's answer, would have called it a match on the third link and responded with, "I see."

In general, the user is forced to make a predetermined response to questions whose alternative links contain no asterisks. He always knows what his options are in this situation, since an undesired reply elicts the list of desired responses. On the other hand, questions whose alternative links do contain an asterisk are never repeated. The user's first reply is always mapped into one of the alternatives.

A certain amount of care must be taken in the construction of a reactive message to keep the user from being forced down a path he doesn't really want to take. The saving factor here is that all the responses of the user are being saved by the agent for transmittal back to the sender. So if the user replies "yes I do," and the agent is looking for "yes," the reply matches but the entire sentence is transmitted back to the sender, not just the "yes." Thus when the user cannot find an alternative he likes, he can pick one anyway but qualify his reply with extra comments.

Design Considerations

The current design of the reactive message is rather unsophisticated. There are a number of ways in which it could be extended to make it a more effective tool for communication. Each extension, however, raises a basic design issue which must be faced.

Exponential Growth. If the data base is tree-structured, its size grows exponentially as questions are added. One way around this problem is to organize the data base as a net rather than a tree. Then interconnections between arbitrary nodes eliminate redundancy and thus reduce the size of the data base. In the current implementation, arbitrary nets are not permitted but some interconnection of nodes is allowed.

Natural-Language Processing. The reactive message should be a short, yet powerful program or agent for eliciting information from a user. These two goals are not really compatible, however, as natural-language processing is a formidable task. One problem is synonym recognition. If the user answers "OK" instead of "yes," the agent should be able to recognize this as an equivalent reply. Two immediate issues arise: (1) Should the sender or the message itself take responsibility for defining the synonym classes? (2) Once the classes are defined, how should the message go about mapping the user's reply into one of the classes? This issue is avoided in the current implementation by not allowing synonym recognition.

Another problem that confounds the synonym problem is that of recognizing logical combinations of replies within a single response. For example, the sender might want to specify that a particular alternative link should match only if the user's response contains either a "yes" or a "no," or should match if it contains both a digit and "morning" but not if it also contains an "except." Logical combinations are not allowed in the current implementation.

Back-up. The recipient of a reactive message doesn't really know where he is going until he gets there, i.e., he can't look ahead to see what's coming. Consequently, at some point in the interaction he may decide he made a mistake in a previous answer and want to change the answer and start fresh from that point. Or he may want to start over from the beginning, or even abort the message in such a way that it is terminated and nothing is sent back to the sender. Such a capability, though sorely needed, will greatly increase the complexity of the program. This issue has not been addressed in the current implementation.

User-Originated Questions. It would be useful to have the message capable of answering questions as well as asking them. This would require some rather complex natural-language processing capabilities, however, which would probably be unjustifiably costly.

Machine-Readable Output. Besides transmitting back to the sender the complete protocol of the user-agent interaction, a mapping of the user replies onto the

expected replies should also be sent. This normalization of the user response would make it relatively easy for another agent to analyze the response of the user and make decisions based on that response.

Creation of Reactive Messages

The reactive message is a fairly complicated program that talks to the user and records his answers. Writing such programs can be a very tedious and time-consuming task, one that requires remembering exactly where you are at all times in the tree of questions being generated. To make this job easier, a second RITA agent called WRITER has been designed to help the sender create his reactive message. WRITER queries the sender, uses his replies to construct the message, and automatically sends the message to the user (recipient) along with a piece of standard computer mail telling him he has a message. The message itself does the rest; that is, it queries the user, records his responses (plus a complete record of the interaction) on a file owned by the sender, and tells the sender that the transaction has taken place.

The protocol of the agent-sender interaction that led to the creation of the reactive message diagrammed earlier is shown below:

AGENT: What is your name?
SENDER: don

AGENT: To whom are you sending this message?
SENDER: Professor Smith

AGENT: If the text is :
Dear Professor Smith:
and the reply is "[none]"
then what is the new text?
SENDER: I'm currently trying to organi: a
weekly seminar on message-handling
systems. The only available day of
the week is Tuesday.

AGENT: What are the possible replies?
SENDER: [none]

AGENT: If the text is :
I'm currently trying to organize a
weekly seminar on message-handling
systems. The only available day of
the week is Tuesday.
and the reply is "[none]"
then what is the new text?

SENDER: Would you be interested in
attending such a series of
seminars?

AGENT: What are the possible replies?
SENDER: yes
no

AGENT: If the text is :
Would you be interested in
attending such a series of
seminars?
and the reply is "yes"
then what is the new text?
SENDER: Good

AGENT: What are the possible replies?
SENDER: [none]

AGENT: If the text is :
Would you be interested in
attending such a series of
seminars?
and the reply is "no"
then what is the new text?
SENDER: Why not?

AGENT: What are the possible replies?
SENDER:

AGENT: If the text is :
Good
and the reply is "[none]"
then what is the new text?
SENDER: What time, between 11am
and 4pm, is best for you?

AGENT: What are the possible replies?
SENDER: 12
1

AGENT: If the text is :
Why not?
and the reply is " "
then what is the new text?
SENDER: OK, thanks anyway
for your response.

AGENT: What are the possible replies?
SENDER: [none]

AGENT: If the text is :
What time, between 11am
and 4pm, is best for you?
and the reply is "12"
then what is the new text?
SENDER: Noon is a bad
time for most
people.

AGENT: What are the possible replies?
SENDER: [none]

AGENT: If the text is :
What time, between 11am
and 4pm, is best for you?
and the reply is "1"
then what is the new text?
SENDER: That sounds
good to me.

AGENT: What are the possible replies?
SENDER: [merge]

AGENT: If the text is :
What time, between 11am
and 4pm, is best for you?
and the reply is " "
then what is the new text?
SENDER: I see.

AGENT: What are the possible replies?
SENDER: [merge]

AGENT: If the text is :
OK, thanks anyway
for your response.
and the reply is "[none]"
then what is the new text?
SENDER: Goodbye

AGENT: What are the possible replies?
SENDER:

AGENT: If the text is :
Noon is a bad
time for most
people.

and the reply is "[none]"
then what is the new text?
SENDER: What's your
second choice?

AGENT: What are the possible replies?
SENDER: .

AGENT: If the text is :
What's your
second choice?
and the reply is " "
then what is the new text?
SENDER: Thanks, I'll get back to
you when the responses to
this message are tabulated

AGENT: What are the possible replies?
SENDER: .

AGENT: Who else are you sending this message to?
SENDER: nobody

AGENT: Finished

Note that the sender specifies the expected replies by typing one reply per line and ending this sequence of lines with a period. In place of a reply, he can also type *"none"* to indicate an associative link to the next piece of text, *"merge"* to indicate that the link being created is to be merged with the one that was just created, or just a carriage return to indicate that the link being created will match any reply of the recipient.

There are a number of problems involved in implementing a message-writing agent. The main problem is in designing the message to take the form of a general net structure but still be easy for the sender to create. The net structure, rather than a restrictive tree structure, is needed to avoid repetition of similar substructures and to curb the exponential growth of the data base. In the current implementation, only merging is permitted, which transforms the tree into a special type of net. It is clear that simple ways of specifying general interconnections in the net are needed.

Another problem is putting the user into context during the user-agent dialogue that results in the creation of the message. The current approach is to have the agent repeat the last piece of text and the reply for the branch being extended, as shown below:

AGENT: If the text is :
What time, between 11am
and 4pm, is best for you?
and the reply is "1"
then what is the new text?

This, combined with the breadth-first generation of the tree, leads to a dialogue that brings the user into context with a minimum of confusion. A better approach would be to display the entire tree (in some abbreviated form) each time the user is to enter new text and have him decide which branches should be extended next. He could indicate his choice by pointing with a light pen or moving a cursor to the appropriate spot. If the message being created has a very complex structure, the user may find it necessary to sketch a diagram or flow chart illustrating where the questions are to be asked and what rules are to be expected for each one. In such a situation, a flow chart would help keep the user from becoming confused or lost during message creation.

V. CONCLUSIONS

The work that has been performed to date in knowledge acquisition using RITA agents has a number of implications for knowledge-based systems. First, the production-system framework can be used for implementing dynamic programs, programs that create new data and code. In particular, the RITA architecture is such that new RITA agents can be created relatively easily through combined user-agent interactions. Here, the production system formalism is the critical factor when it comes to the representation of the program being created. That is, the success we have obtained with RITA agents creating other RITA agents is due to the fact that the programs being created—TASK, TEACH, TRANSFER, and MESSAGE—are all based on the RITA production-system architecture.

Second, it has been demonstrated that user agents can be applied to the problem of program creation and can produce programs that have some practical application. The TASK agent produced by EP-1 is a viable, general program for file transfer. EP-1 has also been used to create agents that access the New York Times Information Bank and retrieve abstracts of news articles pertinent to the user's current interests. The TEACH agent provides a way to tutor users without modifying the systems the users have to interact with, and has been used to teach naive users about RITA. The TRANSFER agent provides a way of automatically acquiring large amounts of data about the use of the ARPAnet, in a form that can be applied directly to the problem of file transfer. The MESSAGE agents created by WRITER can be used for personal communication between computer users but are particularly useful as data-gathering tools, i.e., as interactive questionnaires.

Finally, the potential that exemplary programming has for man-machine interface applications has been demonstrated. It is a first step in helping the naive user create programs without having to learn a programming language or other artificial language for stating the problem to be solved. This technique is effective for programs that involve much repetition of similar sequences of processing with few dynamic variables, such as man-machine interface programs. The crux of the learning problem in exemplary programming is how to generalize a program after seeing only one or two paths to the solution, i.e., one or two examples of how the task can be accomplished. We believe that this important problem is solvable and that considerable time and effort should be devoted to its solution.

Appendix A

A RITA AGENT THAT TRANSFERS A FILE FROM AN ARBITRARY ARPANET SITE TO RAND-UNIX

```
[OBJECTS:]

OBJECT agent<1>: ;

OBJECT current-system<1>: ;

OBJECT system<1>:
        response    IS  " ",
        input       IS  " ",
        count       IS  "0";

[RULES:]

        RULE 1:
            IF: the prompt OF the agent IS KNOWN
          THEN: SEND the prompt OF the agent TO user
                & RECEIVE the NEXT {ANYTHING 'line-contents'
                  FOLLOWED BY "^j"}
                  FOR 15 SECONDS FROM the user
                & SET the new-reply OF the agent TO 'line-contents'
                & SET the prompt OF the agent TO NOT KNOWN;

        RULE 2:
            IF: the new-reply OF the agent IS KNOWN
                & the new-reply OF the agent IS NOT " "
          THEN: SET the reply OF the agent TO the new-reply OF the agent
                & SET the new-reply OF the agent TO " ";

        RULE 3:
            IF: the reply OF the agent IS KNOWN
          THEN: SEND the reply OF the agent TO system
                & SEND " " TO user
                & SEND concat( "sent: ", the reply OF the agent ) TO user
                & SET the reply OF the agent TO NOT KNOWN;

        RULE 4:
            IF: the name OF the current-system IS NOT KNOWN
                & the state OF the agent IS NOT KNOWN
                & the response OF the system CONTAINS {" "}
          THEN: SET the name OF the current-system TO "file transfer
                  program"
                & SET the state OF the agent TO "give the host name"
                & SET the response OF the system TO " "
                & SET the reply OF the agent TO "ftp";
```

```
RULE 5:
     IF: the name OF the current-system IS "file transfer program"
         & the state OF the agent IS "give the host name"
         & the response OF the system CONTAINS {"Host: "}
   THEN: SET the state OF the agent TO "login"
         & SET the response OF the system TO " "
         & SET the reply OF the agent TO "cmu-10a"
         & SET the prompt OF the agent TO
           "What system shall I retrieve the file from?...~$";

RULE 6:
     IF: the state OF the agent IS "login"
         & the response OF the system CONTAINS {"> "}
         THEN: SET the response OF the system TO " "
         & SET the reply OF the agent TO "user";

RULE 7:
     IF: the response OF the system CONTAINS {"username: "}
   THEN: SET the state OF the agent TO "give password"
         & SET the response OF the system TO " "
         & SET the reply OF the agent TO "a330dw28"
         & SET the prompt OF the agent TO
           "What is your user name on this system?...~$";

RULE 8:
     IF: the state OF the agent IS "give password"
         & the response OF the system CONTAINS {"> "}
   THEN: SET the response OF the system TO " "
         & SET the reply OF the agent TO "pass";

RULE 9:
     IF: the response OF the system CONTAINS {"Password: "}
   THEN: SET the state OF the agent TO "retrieve the file"
         & SET the response OF the system TO " "
         & SET the reply OF the agent TO "abcdef"
         & SET the prompt OF the agent TO
           "What is your password on this system?...~$";

RULE 10:
     IF: the state OF the agent IS "retrieve the file"
         & the response OF the system CONTAINS {"> "}
   THEN: SET the response OF the system TO " "
         & SET the reply OF the agent TO "retr";

RULE 11:
     IF: the response OF the system CONTAINS {"remotefile: "}
   THEN: SET the response OF the system TO " "
         & SET the reply OF the agent TO "mail.boxa330dw28"
         & SET the prompt OF the agent TO
           "What file do you want retrieved?...~$";
```

```
RULE 12:
      IF: the response OF the system CONTAINS {"localfile: "}
    THEN: SET the state OF the agent TO "say goodby to the remote
            host"
          & SET the response OF the system TO " "
          & SET the reply OF the agent TO "newdata"
          & SET the prompt OF the agent TO
            "What do you want to call the retrieved file?...~$";

RULE 13:
      IF: the state OF the agent IS "say goodby to the remote host"
          & the response OF the system CONTAINS {"> "}
    THEN: SET the name OF the current-system TO "unix"
          & SET the state OF the agent TO "quit"
          & SET the response OF the system TO " "
          & SET the reply OF the agent TO "bye";

RULE 14:
          IF: the name OF the current-system IS "unix"
          & the state OF the agent IS "quit"
          & the response OF the system CONTAINS {"% "}
    THEN: SEND "the file transfer has been completed" TO user
          & RETURN SUCCESS;

RULE 15:
      IF: the input OF the system IS " "
          & the count OF the system IS less THAN 20
    THEN: RECEIVE FOR 5 SECONDS FROM system AS the input OF
            the system
          & SET the count OF the system TO 1 + the count OF the
            system;

RULE 16:
      IF: the count OF the system IS less THAN 20
    THEN: SET the response OF the system TO
            concat( the response OF the system, the input OF the
            system)
          & SET the input OF the system TO " "
          & SET the count OF the system TO 0
          & SEND concat( "got: ", the response OF the system ) TO
            user;
```

Appendix B

A PROTOCOL OF A USER RUNNING THE TASK AGENT

User input is shown in italics, TASK agent response in normal type, and system response in boldface.

% rita go

go:

sent: ftp
got: Host:
What system shall I retrieve the file from?. . .*sri-ai*

sent: sri-ai
got: Connections established.

> > 300 SRI-AI FTP Server 1.44.0.0 - at THU 20-MAY-76 10:21-PDT

sent: user
got: ?Command argument too long
> >

sent: user
got: username:
What is your user name on this system?. . .*rand*

sent: rand
got: 330 User name accepted. Password, please.
>

sent: pass
got: Password:
What is your password on this system?. . .*abcdef*

sent: abcdef
got:
230 Login completed.
>

sent: retr
got: remotefile:
What file do you want retrieved?. . .*test.bas*

sent: test.bas
got: ?Command argument too long
> >

```
sent: retr
got:  ?Command argument too long
> >

sent: retr
got:  ?Command argument too long
> >

sent: retr
got:  remotefile:
What file do you want retrieved?. . .test.bas

sent: test.bas
got:  localfile:
What do you want to call the retrieved file?. . .
sent: newdata
got:  255 SOCK 3276932611

got:  255 SOCK 3276932611
250 ASCII retrieve of <RAND>TEST.BAS;1 started.

got:  255 SOCK 3276932611
250 ASCII retrieve of <RAND>TEST.BAS;1 started.
252 Transfer completed.
>

sent: bye
got:  231 BYE command received.
%
the file transfer has been completed

* exit;
exiting.
```

Appendix C

A PROTOCOL OF A USER-TEACH INTERACTION

User input is shown in italics, TEACH agent response in normal type, and system response in boldface.

% rita use.teach

use.teach:

This is a RITA ruleset designed to help you learn
to use RITA. Type "help;" if you need help.

[End all RITA commands with a semicolon(;)]

* *help;*

In addition to the standard RITA input, you may also type *show me*, followed by either *commands, actions, functions, rules, goals*, (any command name), (any action name), or (any function name). The result will be either text or an on-line demo. Each time you repeat the *show me* request, you will be given a new demo.

Typing *again;* is equivalent to typing the last *show me* command.

* *show me commands;*

continue
edit
exit
load
news
quiet
run
scratch
set ordered
set trace
set unordered
shell
verbose
(any RITA action)
(any rule body)

* *show me actions;*

create
deduce
delete

display
put
receive
remove
return
send
set
stop at
trace
unstop
untrace
(any function call)

* *show me functions;*

clock
concat
delay
div
index
minus
nsubstr
plus
substr
times

* *show me Los Angeles;*
I don't know about Los Angeles;

* *show me deduce;*

[start of deduce demo]

* object book;
[Object book<1> added]

* display object book;
OBJECT book<1>: ;

* deduce the name of the book;
What is the name of the book?
Response: Gone with the Wind

* display object book;
OBJECT book<1>:
name IS "Gone with the Wind";

```
* delete object book;

[end of demo: 2 deduce demo's available]

* again;

[start of deduce demo]

* object hat;
[Object hat<1> added]

* goal demo
  if:  the color of the hat is "brown"
  then: set the owner of the hat to "Richard";
[Goal demo added]

* deduce the owner of the hat;
What is the color of the hat
        Response:brown

* display object hat;
OBJECT hat<1>:
        owner   IS   "Richard",
        color   IS   "brown";


* delete object hat; delete goal demo;

*
[end of demo: 2 deduce demo's available]

* show me trace;

[start of trace demo]

* rule test1
  if: value of count is greater than 4
  then: send "The value of count is 5." to the user
        and return success;
[Rule test1 added]

* rule test2
  if: value of count is less than 5
  then: set the value of count to 1 + value of count;
[Rule test2 added]

* object count: value is 0;
[Object count<1> added]
```

* trace all rules;

* run;

------> rule test2 about to fire
------> rule test2 about to fire
------> rule test2 about to fire
------> rule test2 about to fire
------> rule test2 about to fire
------> rule test1 about to fire
The value of count is 5.
Success!

* delete object count;

* untrace all rules; delete rule test1; delete rule test2;

*
*
[end of demo: 2 trace demo's available]

* *send "this is the user talking" to the user;*
this is the user talking

* *rule 1 if value of c is less than 5*
then send value of c to user
and set value of c to value of c + 1;
[Rule 1 added]

* *run;*
Failure.

* *object c value is 0;*
[Object c<1> added]

* *run;*
0
1
2
3
4
Failure.

* *show me plus;*

[start of plus demo]

* send plus(3,4) to the user;
7

```
* send 3 + 4 to the user;
7

* object people: number is 8;
[Object people<1> added]

* send plus(plus(2,7), number of people) to the user;
17

* delete object people;

[end of demo: 2 plus demo's available]

* exit;
exiting.
%
```

Appendix D

A PROTOCOL OF A USER-TUTOR INTERACTION TO CREATE A PROGRAM TO TEACH LISP

User input is shown in italics, TUTOR agent response in normal type, and system response in boldface.

% rita tutor
tutor:
What is the name of the system to be demonstrated?
lisp
What prompt character does lisp use?
->
Please type a command to call the system to be demonstrated
% sys
Type text that would be appropriate as a
response to a query for help in the lisp system.
Terminate the text with double <carriage return>.
Type "show me" followed by "functions" or a
name of a function to elicit a list of function
names or a demonstration of the function.

Classify the types of capabilities you intend to
demonstrate in lisp. On one line put the category name
and on the next the list of items in that category, e.g.:

commands
load news run exit.
actions
put set send remove.

terminating the text by typing a line
containing just a <carriage return>.
functions
car
setq
.

Please give a name for each demo when asked. A demo
can consist of several examples. To indicate the end
of an example type "(end of example)". To indicate
the end of all the examples of that particular demo,
type "(end of demo)".

What are you going to demonstrate? *car*
–> *(car '(a b c d))*
a
–>*(car '(this is a test))*
this
–>*(car '((a b)(c d)))*
(a b)
–>*(end of example)*
–>*(end of demo)*

What are you going to demonstrate? *setq*
–> *(setq a 12)*
12
–>*a*
12
–>*(plus a 13)*
25
–>*(setq b '(cdef))*
(c d e f)
–>*(cons a b)*
(12 c d e f)
–>*(end of example)*
–> *(setq a 'first)*
first
–>*(setq b '(second third))*
(second third)
–>*(setq a (cons a b))*
(first second third)
–>*a*
(first second third)
–>*(end of example)*
–>*(end of demo)*

What are you going to demonstrate? *(nothing)*

Type a command to exit the system being demonstrated
–> *(exit)*
exiting.
%

Appendix E

A PROTOCOL OF THE USE OF THE TEACH AGENT CREATED FOR LISP*

User input is shown in italics, TEACH agent response in normal type, and system response in boldface.

% rita use.teach.t

use.teach.t:

This is a program designed to help you learn
to use lisp. Type 'help' if you need help.

-> help
Type *show me*, followed by *functions* or a
name of a function to elicit a list of function
names or a demonstration of the function.

-> show me functions
car
setq

-> show me car

[start of car demo]

-> (car '(a b c d))
a
->(car '(this is a test))
this
->(car '((a b)(c d)))
(a b)
[end of demo: 1 car demo(s) available]

-> show me setg
I don't know about setg

-> show me setq

[start of setq demo]

*See App. D.

->(setq a 12)
12
->a
12
->(plus a 13)
25
->(setq b '(c d e f))
(c d e f)
->(cons a b)
(12 c d e f)
[end of demo: 2 setq demo(s) available]

-> *again*

[start of setq demo]

->(setq a 'first)
first
->(setq b '(second third))
(second third)
->(setq a (cons a b))
(first second third)
->a
(first second third)
[end of demo: 2 setq demo(s) available]

-> *(setq r '(this is a sentence))*
(this is a sentence)
->*(car r)*
this
->*(exit)*

exiting.
%

BIBLIOGRAPHY

Anderson, R. H., "The Use of Production Systems in RITA To Construct Personal Computer 'Agents,'" *Proceedings of the Workshop on Pattern-Directed Inference Systems*, SIGART Newsletter No. 63, 1977, 23-28.(a)

Anderson, R. H., M. Gallegos, J. J. Gillogly, R. B. Greenberg, and R. Villanueva, *RITA Reference Manual*, The Rand Corporation, R-1808-ARPA, 1977 (b).

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

Balzer, R. M., *Automatic Programming*, Institute Technical Memorandum, USC/ Information Sciences Institute, Los Angeles, 1972.

Balzer, R. M., "A Global View of Automatic Programming," *Proceedings of the Third International Conference on Artificial Intelligence*, Stanford, California, 1973, pp. 494-499.

Biermann, A. W., *Regular LISP Programs and Their Automatic Synthesis from Examples*, Computer Science Department Report CS-1976-12, Duke University, 1976.

Biermann, A. W., and R. Krishnaswamy, *Constructing Programs from Example Computations*, Computer and Information Science Research Center Report CISRC-TR-74-5, Ohio State University, 1974.

Bobrow, D., and T. Winograd, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, Vol. 1, 1976, pp. 3-46.

Brooks, R., "Production Systems as Control Structures for Programming Languages," *Proceedings of the Workshop on Pattern-Directed Inference Systems*, SIGART Newsletter, No. 63, 1977, pp. 33-37.

Buchanan, B. G., and N. S. Sridharan, "Rule Formation on Non-homogeneous Classes of Objects," *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, California, 1973.

Buchanan, J. R., "A Study in Automatic Programming," *Computer Science Report*, Carnegie-Mellon University, 1974.

Davis, R., B. Buchanan, and E. Shortliffe, *Production Rules as a Representation for a Knowledge-Based Consultation Program*, Stanford University, Artificial Intelligence Laboratory, Memo AIM-266, 1975.

Davis, R., *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*, Stanford University, Artificial Intelligence Laboratory, Memo AIM-283, 1976.

Duda, R.O., P. E. Hart, and J. Nils Nilsson, *Subjective Bayesian Methods for Rule-Based Inference Systems*, Stanford Research Institute, SRI Technical Note 124, 1976.

Galler, B., and A. Perlis, *A View of Programming Languages*, Addison-Wesley, 1970.

Goldberg, P. C., "Automatic Programming," *Programming Methodology*, G. Goos and J. Hartmanis (eds.), "Lecture Notes," *Computer Science*, Vol. 23, Springer-Verlag, New York, 1975.

Green, C., *The Design of the PSI Program Synthesis System*, Second International Conference on Software Engineering, San Francisco, California, 1976, pp. 4-18.

Green, C., and D. Barstow, "Some Rules for the Automatic Synthesis of Programs," *Proceedings of the Fourth International Conference on Artificial Intelligence*, 1975, pp. 232-239.

Green, C., J. Waldinger, R. Barstow, D. Lenat, B. McCune, D. Shaw, and L. Steinberg, *Progress Report on Program-Understanding Systems*, Stanford University, Artificial Intelligence Laboratory, Memo AIM-240, 1974.

Hewitt, C., *Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in Robots*, TR-258, Ph.D. thesis, MIT Artificial Intelligence Laboratory, 1972.

Klahr, P., *The Deductive Pathfinder: Creating Derivation Plans for Inferential Question-Answering*, System Development Corporation, SP-3842, 1975.

Lenat, D., "Beings: Knowledge as Interacting Experts," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 126-133.

Lenat, D., *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*, Stanford University, Artificial Intelligence Laboratory, Memo AIM-286, 1976.

Manna, Z., and R. J. Waldinger, "Knowledge and Reasoning in Program Synthesis," *Artificial Intelligence*, 1975, Vol. 6, pp. 175-208.

Markov, A. A., *Theory of Algorithms*, National Academy of Sciences, USSR, 1954.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*, The MIT Press, 1965.

Newell, A., "A Theoretical Exploration of Mechanisms for Coding the Stimulus," *Coding Processes in Human Memory*, A. W. Melton and E. Martin (eds.), Winston and Sons, Washington, D.C., 1972.

Newell, A., "Production Systems: Models of Control Structures," *Visual Information Processing*, W. C. Chase (ed.), Academic Press, New York, 1973.

Newell, A., and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

Newell, A., and J. McDermott, *PSG Manual, System Version PSG2*, Carnegie-Mellon University, 1974.

Post, E. L., "Formal Reductions of the General Combinatorial Decision Problem," *American Journal of Mathematics*, Vol. 65, 1943, pp. 197-268.

Rulifson, J. F., J. A. Derksen, and R. J. Waldinger, *QA4: A Procedural Calculus for Intuitive Reasoning*, Stanford Research Institute, Menlo Park, 1972.

Rychener, M. D., *The Student Production System: A Study of Encoding Knowledge in Production Systems*, Department of Computer Science, Carnegie-Mellon University, 1975.

Rychener, M. D., *Introduction to Psnlst*, Department of Computer Science, Carnegie-Mellon University, 1976.

Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, Vol. 2 of the Artificial Intelligence Series, 1976.

Siklossy, L., and D. A. Sykes, "Automatic Program Synthesis from Example," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 268-273.

Standish, T. A., "Scenarios for Use of an Intelligent Terminal," University of California, Irvine, unpublished manuscript.

Sussman, G. J., and D. V. McDermott, *Why Conniving is Better than Planning*, MIT Artificial Intelligence Laboratory, Memo 255A, 1972.

Teitelman, W., *INTERLISP Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1975.

Vere, S. A., *Relational Production Systems*, Department of Information Engineering, University of Illinois, 1975.

Waterman, D. A., "Generalization Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, Vol. 1, 1970, pp. 121-170.

Waterman, D. A., "Adaptive Production Systems," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp. 296-303.

Waterman, D. A., "Serial Pattern Acquisition: A Production System Approach," *Pattern Recognition and Artificial Intelligence*, C. H. Chen (ed.), Academic Press, New York, 1976, pp. 529-553 (a).

Waterman, D. A., *An Introduction to Production Systems*, The Rand Corporation, P-5751, 1976 (b).

Waterman, D. A., and F. Hayes-Roth, "An Overview of Pattern-Directed Inference Systems," *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Academic Press, New York, 1978.

Waterman, D. A., and A. Newell, "PAS-II: An Interactive Task-Free Version of an Automatic Protocol Analysis System," *IEEE Transactions on Computers*, C-25, 1976.