A RULE-BASED APPROACH TO KNOWLEDGE ACQUISITION
FOR MAN-MACHINE INTERFACE PROGRAMS

D. A. Waterman

June 1978

P-5823
The Rand Paper Series

Papers are issued by The Rand Corporation as a service to its professional staff. Their purpose is to facilitate the exchange of ideas among those who share the author's research interests; Papers are not reports prepared in fulfillment of Rand's contracts or grants. Views expressed in a Paper are the author's own, and are not necessarily shared by Rand or its research sponsors.

The Rand Corporation
Santa Monica, California 90406
ABSTRACT

The development of computer programs, called agents, that act as man-machine interfaces for computer users are described. 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, or "personal computer agents," are divided into two main categories: those that interface the user to computer systems he wishes to use and those that interact with the user to acquire the knowledge needed to create these interface programs. The relationship between the interface program and the knowledge acquisition program is that of parent-offspring. Three types of parent-offspring RITA agent pairs 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 task agent) to perform the same task, 2) 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, and 3) a reactive-message creating agent which elicits text from a user (the sender) and from it creates a new RITA agent which is a reactive message. The reactive message is 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.
ACKNOWLEDGEMENTS

The comments and criticisms of Bob Anderson and other members of the Information Sciences Department staff at Rand are gratefully acknowledged.
I. INTRODUCTION

The rapid advance in the development of minicomputer hardware makes it likely that computers will soon be available to a large class of users as personal tools for scientific and business applications. Each user will have, in essence, his own personal computer, whether it be a minicomputer inside a terminal or a slice of a remote timesharing system. The typical user of a personal computer will have little desire to learn programming or complex protocols for communicating with computer operating systems. Instead, he will want to use the computer as a tool, one that is easy to learn but powerful enough to provide a variety of resources to aid him in his daily work. The basis for this type of tool is a software system that can react intelligently to the individual needs of the user.

The RITA system has been developed to help provide such a tool. RITA is a rule-based software system that has been applied to both the man-machine interface problem (Anderson, 1977; Waterman, 1978a) and the heuristic modeling problem (Waterman and Jenkins, 1977), i.e., representing the knowledge of an expert in an information processing model designed to aid users in on-line decision making. In this paper the use of RITA to write programs that interface the user to local and external computer systems is explored in detail. This type of program is called an agent to emphasize its role as an intelligent assistant to the user, one which may either interact heavily with the user or run autonomously, even to the extent of
initiating and terminating its own operation.

A RITA rule has the form "IF premises THEN actions," and a collection of rules constitutes a rule set. The production system control structure provides the simplicity and modularity needed to make program organization straightforward and program modification relatively easy. The use of a conversational syntax makes RITA programs easy to read and understand compared to conventional programming languages; thus, they tend to be self-documenting. Finally, the language primitives in RITA make it quite easy to interact with other computer systems, including initiating and running several jobs in parallel on external systems.

To illustrate the RITA architecture consider the simplified mail retrieval agent shown in Figure 1. The data base consists of a set of unordered object-attribute-value triples, in this case just one object called "system" whose attribute is "response" and whose value is not known. The rule set consists of three rules whose left-hand sides are premises evaluated in terms of the current data base and whose right-hand sides are actions that either change the contents of the data base or initiate input/output operations. The control structure is that of a conventional "rule ordered" production system (Davis and King, 1975; Waterman, 1975, 1976a, 1976b). Thus, a control cycle consists of searching the rules, from top to bottom, to find one whose premises are true, and then executing the actions associated with that rule. After a rule is executed
the search for a "true" rule starts over again from the top.

OBJECTS:

OBJECT system<1>:
  response IS NOT KNOWN;

RULES:

RULE 1:
  IF: the response OF the system IS NOT KNOWN
  THEN: SEND "ftp cmu-10a" TO system
        & RECEIVE NEXT {">"} FROM system
        AS the response OF the system;

RULE 2:
  IF: the response OF the system CONTAINS
     {"Connections established"}
  THEN: SEND "retr mail[a330dew28] newmail" TO system
        & RECEIVE NEXT {"> "} FROM system
        AS the response OF the system;

RULE 3:
  IF: the response OF the system CONTAINS
     {"Transfer completed"}
  THEN: SEND "bye" TO system
        & RECEIVE NEXT {">"} FROM system
        AS the response OF the system
        & SEND "CMU mail is now in file newmail." TO user
        & RETURN SUCCESS;

Figure 1. A Simple RITA Agent for Retrieving
Mail from the CMU-10A System
(Reserved words are shown in upper case)
Rule 1 of the mail agent sends a command to the local operating system requesting a connection to cmu-10a for the purpose of file transfer. It then waits until it receives a response containing the appropriate prompt character before it relinquishes control to the RITA interpreter. Rule 2 fires only if the response of the external system indicates that a connection to the desired host has been made. It then initiates the retrieval by sending the retrieval request to the external system. It also waits for a response containing an appropriate prompt before terminating rule execution. Rule 3 fires when the response of the external system indicates that the transfer has been successfully completed. It then closes the connections to the host and terminates execution of the agent (the "return success" action) after notifying the user that the mail file has been retrieved.

If the agent illustrated in Figure 1 is augmented with output statements of the form:

```plaintext
& send "SENT:   ftp cmu-10a" to user
& send concat("RECEIVED: ", response of system) to user
```

it will provide the user with a trace indicating what has been sent to the external system and what has been received in reply.

Figure 2 shows such a trace as the agent retrieves file "mail" from Carnegie-Mellon University into a local file called "newmail."
SENT: ftp cmu-10a
RECEIVED: % Connections established.

300 CMU10A 8.3/DEC 6.02VM FTP Server 4(31)
>
SENT: retr mail[a330dw28] newmail
RECEIVED: 050 Other jobs same PPN
1753 29-Jun-77 Wed
...SYS:NEWS(5-13)
Hydra: 10-Jun-77 New terminal interface;
see TMUX.DOC[N811HY97]
255 SOCK 12033
250 RETR started.
252 Transfer completed.
>
SENT: bye
RECEIVED: 050 Other jobs same PPN
Job 23, User [N900AR00]
Logged off TTY146 1753 29-Jun-77
Another job still logged in under [N900AR00]
Runtime 0 Min, 00.60 sec; Kilocore sec: 1
Connect time 0 hr, 0 min, 12 sec;
Total charge: $0.03
231 Bye.
%

CMU mail is now in file newmail.

Figure 2. Trace Produced by the Mail Retrieval Agent
(see Figure 1).

The mail agent of Figure 1 was written solely to
illustrate some of the basic features of the RITA architecture,
and although it will successfully retrieve mail it is not a
representative file retrieval agent in terms of size and
complexity. A truly useful file retrieval agent would need a
number of features not included in the mail agent, such as the
ability to interact with the user and to respond appropriately to
unexpected ARPAnet error messages (see Waterman, 1978a, 1978b).
This paper discusses a number of applications of the RITA architecture to man-machine interface problems within the context of knowledge acquisition. Section II defines and describes user agents and categorizes them according to both their knowledge acquisition capability and how they relate to the user. Section III discusses the use of agents for knowledge acquisition tasks. To illustrate acquisition of procedural knowledge, an exemplary programming agent called EP-1 is described. This agent monitors the interaction between a user and an external system, queries the user about the task being performed, and on the basis of the information thus gained creates a program called a task agent which can perform the task. To illustrate the acquisition of declarative information two agents are described, a tutor agent and a writer agent. The tutor agent watches an expert demonstrate various features of some local interactive system and then generates a program called a teach agent which can henceforth be called upon by novice users of the system to help teach them system features. The writer agent elicits from the user a specification of a dialogue that he would like to carry on with another user and converts this specification into a program called a reactive message: an agent that is able to carry on a dialogue with some recipient and then relay the result of the conversation back to the sender of the message. The conclusions are presented in Section IV.
II. USER AGENTS

A user agent is a small program that sits between the user and the system he is interacting with and is capable of performing a variety of tasks for the user. Programs of this sort, written in the RITA (Rule-Directed Interactive Transaction Agent) system, are sometimes referred to as interactive transaction (IT) agents. User agents interface the user to external systems, i.e., they provide him 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 newspaper articles. An agent interfacing a user to the NYTIB could query the user about what kinds of abstracts he wanted to retrieve and then access the NYTIB and perform the retrieval, freeing the user to perform other tasks. Alternatively, it could simply pass messages back and forth between the user and the system, answering all the queries from the NYTIB that it could while mapping the queries it couldn't answer into a form intelligible to the user for him to answer 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 agent which does not mask any of the characteristics of the system. If he only wants to expedite the interaction then an agent that hides or simplifies the system characteristics is usually appropriate.
User agents are also useful for message handling tasks (Anderson, 1977). This includes reading and analyzing incoming mail and informing the user when important or high priority messages arrive. Agents can also be designed to initiate mail under certain circumstances and even respond to the replies that are elicited, e.g., if the agent were trying to use a local system, such as a network control program, and received error messages when attempting to access it, the agent could send a message to the person responsible for maintaining the system asking for help or maintenance. Sophisticated agents have been designed which query the user to obtain information needed to create reactive messages (see section III), i.e., messages in program form which carry on a dialogue with the message recipient.

Agents are particularly useful for providing mundane, periodic services such as network accessing, secretarial, 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 accounting.

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

User agents can be categorized according to how they relate to the user and what sort of mechanism they have for acquiring knowledge. If the agent relates to the user in a very direct way, it can be considered an active agent, but if it relates in a somewhat indirect way it is more appropriately labeled a passive agent. Neither of these classifications is absolute since actual agents can exhibit both active and passive properties to varying degrees. Figure 3 shows a diagram of an active agent.

![Figure 3. Active User Agent](image-url)
An active agent stands between the user and the external system, preventing the user from talking directly to the system and thus hiding many of the characteristics of that system. The agent may query the user, translate his response into something the external system will understand, and immediately send it to the system. Or it may extract the information needed to perform some task for the user, process it, but not perform the task until some later time. Of course the agent is not required to interact with the user. Instead it may perform some routine task periodically and leave the result where the user can find it.

In contrast, a passive agent maintains a low profile, giving the user the impression that he is talking directly to the other system. It usually passes the user's input directly to the external system and the reply of the system right back to the user. A diagram of a passive agent is shown in Figure 4.

![Diagram of Passive User Agent](image)

Figure 4. Passive User Agent

The passive agent lets the user take the initiative and guide the course of the interaction, i.e., the user queries the agent, gives commands, and generally maintains control. The
active agent, however, does not often relinquish control to the user, and in the process of trying to accomplish its task usually queries the user when task-specific information is needed.

Agents can also be categorized according to their knowledge acquisition capabilities. If the agent cannot acquire new knowledge it is considered **static**. If it can acquire knowledge and store it for later use in new situations it is considered **dynamic**. The type of dynamic agents we are concerned with here are those which can create, build, or modify other agents. These dynamic agents map the information they receive from the user and external systems into knowledge that is integrated into the agents being created. In RITA this knowledge is either in the form of declarative knowledge (RITA objects) or procedural knowledge (RITA rules). Thus, an agent capable of mapping its knowledge into rules (or both rules and objects) for a new agent would be classified as "procedural." Conversely, an agent capable only of mapping its knowledge into new objects would be classified as a "declarative" agent. Three dynamic RITA agents which have been implemented and the offspring of these agents are summarized in Table I.
<table>
<thead>
<tr>
<th>Name</th>
<th>Task</th>
<th>Relation To User</th>
<th>Knowledge Acquisition</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP-1</td>
<td>create an agent to perform some job for the user</td>
<td>passive/active</td>
<td>procedural/declarative</td>
<td>TASK</td>
</tr>
<tr>
<td>TASK</td>
<td>perform some low level job for user</td>
<td>active</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>TUTOR</td>
<td>modify an agent so it will be capable of tutoring the user</td>
<td>passive</td>
<td>declarative</td>
<td>TEACH</td>
</tr>
<tr>
<td>TEACH</td>
<td>assist the user in learning some external system</td>
<td>passive</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>WRITER</td>
<td>assist the user in creating a reactive message</td>
<td>active</td>
<td>declarative</td>
<td>MESSAGE</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>interact with the recipient of message and send result to user (sender)</td>
<td>active</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

**TABLE I.** RITA Agents
EP-1 watches a user perform a series of operations on the computer and then writes a program, the TASK agent to perform that task. 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 which 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. Note that three of the agents in Table I are creative, that is, capable of producing or modifying offspring (other agents). The other three agents are the offspring produced by the creative agents.
III. KNOWLEDGE ACQUISITION VIA AGENTS

In this section we focus on the distinction between agents which can acquire procedural knowledge and those which can only acquire declarative knowledge. Since a program is usually thought of as procedural knowledge, we are contrasting program writing agents with data acquisition agents. Research on the development of techniques for automatically creating working computer programs has been underway for some time. This research includes program synthesis by selection (Goldberg, 1975), where the user puts together a program from pre-defined modules, 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 analysis (Balzer, 1972, 1973; Buchanan, 1974; Lenat, 1975), where the task is specified interactively or in some domain-dependent language, and program synthesis from example (Biermann and Krishnaswamy, 1974; Biermann, 1976; Siklossey and Sykes, 1975; Green, 1976), where partial states of the process to be implemented are used as the basis for generating the program. However, agents for acquiring procedural knowledge differ from this previous work primarily in two ways: representation and scope of task.

The user agents described in this paper are written in RITA, and thus procedural knowledge in the agents is represented as a collection of rules. When one agent creates an offspring it
does so incrementally, creating a new rule when new knowledge is extracted from the user and the system he is accessing. The rule-based representation is what makes incremental program creation feasible and is one way this work differs from other research in program synthesis.

The offspring of dynamic agents are rule-based programs which have a relatively simple structure but can perform complex tasks related to network or database accessing. Thus, their scope is narrower than that of the more general-purpose programs that are the goal of the program synthesis research. The tasks best suited for the offspring of the dynamic agents are those with a relatively linear or sequential search space. Generally, the solution paths through the space have many states in common, e.g., in ARPA-net file-transfer tasks the common states might include "accessing the ftp program," "logging into the remote host," "initiating a retrieval request," and "exiting."

RULE ACQUISITION

An example of a user agent that can acquire procedural knowledge in rule form is EP-1, an exemplary programming agent (Waterman, 1978a, 1978b). Exemplary programming (EP) is a form of program synthesis based on specifications, in the form of examples, of the task to be programmed (Biermann, 1976; Green, 1976). EP-1 watches a user perform some sequence of operations
at his terminal, queries him about the task, and uses the information gained to write a task agent, a new program that will perform the task the user illustrated. Of course, to be at all useful the exemplary programming agent must have techniques for making generalizations about what the user has done so it can create a task agent that can perform many different variations of the task, not just the sequence of operations it was shown. If the EP agent does not contain domain-specific knowledge regarding invariant features of the task, it must interact with the user to extract this information.

As shown in Table I, EP-1 has both passive and active characteristics. It passively sits between the user and the external system he is accessing, maintaining a low profile. It is careful not to mask any of the characteristics of the external system as it observes the actions of the user and the responses of the system. However, when it needs information about what the user is currently doing, it actively queries him and incorporates this information into the rules being created. EP-1 is also both procedural and declarative, i.e., it creates a task agent containing both the rules and the data base needed to direct the execution of the rules. This task agent may be interactive and require the user to provide it pertinent information as it is running.

EP-1 is designed to operate in two modes, acquisition and training. Acquisition is basically a program-writing mode in which the agent monitors the activity of the user and writes a
program to perform the task demonstrated by the user. Training is a debugging mode in which the user watches EP-1 run the task agent. When the task agent produces an erroneous response, the user tells EP-1 what the correct response is and why. The EP agent may then use this information to correct the error by adding new rules or modifying existing ones (Waterman, 1970; Davis, 1976).

The operation of EP-1 during acquisition consists of two parts: initialization and the basic cycle, as shown in Figure 5. During initialization the user supplies a name for each type of data object relevant to the task. The basic cycle involves EP-1 watching the user interacting with the external system, querying the user, and creating appropriate RITA rules.

INITIALIZATION

* The user supplies a name for each type of data object relevant to this task.

BASIC CYCLE

* The user sends a message to the external system and receives its reply.

* The user is asked for the current values of the data objects he declared relevant during initialization.

* A RITA rule is created. Its premises reflect the values of the data objects before the user sent his last message to the external system. Its actions reflect the content of his last message and its effect on the data objects.

* The user continues, repeating the cycle until he wishes to quit and types "**finished**."

Figure 5. Operation of EP-1 during Acquisition
For a NYTIB retrieval task the user might specify that the following two types of information are relevant.

* the status of the interaction

(a term describing the current status of the program, e.g., "define date range," or "search for abstracts.")

* the value of the response

(the response the user receives from the external system, e.g., "> " from the NYTIB program.)

The user would also specify that "value of the response" is output information from the external system. EP-1 creates a new rule at the juncture of two cycles, using the information gained in the first cycle to define the premises of the rule and the information gained in the second cycle to define the actions. Figure 6 gives an example of the dialogue needed for rule creation in the NYTIB retrieval task.

The NYTIB retrieval task consists of logging in to their computer system, and answering from 10 to 20 queries designed to elicit the information necessary to retrieve and display abstracts of news articles relevant to the topic of interest. In the example of Figure 6 the user had previously indicated that he was interested in the "PL0" and "bombs," and immediately prior to cycle-1 the system had asked him to specify the dates of interest. The user responded with the range April to June, 1976. The system then displayed the current set of active topics and asked for a boolean expression to
define the search. The user specified a search for topic 1 alone and received information regarding the number of relevant abstracts found.

Figure 6. Two Cycles of Dialogue Needed
For Rule Creation in EP-1

The rule created from the two cycles of dialogue in Figure 6 is shown below. Note that the premises are extracted from cycle-1 and the actions from cycle-2.

RULE M:
IF: the status OF the interaction IS "search for abstracts"
& the value OF the response CONTAINS {"> "}
THEN: SET the status OF the interaction TO
"display abstracts on screen"
& SET the value OF the response TO " "
& SET the reply OF the agent TO "b/1";

This rule could be paraphrased as "If it is the time to
search for abstracts and the system (NYTIB) has indicated it is ready by giving the appropriate response (in this case a prompt character) then note that the next step will be to display the abstracts on the screen, set the value of the response to the empty string, and note that the choice "b/1" should be sent to the system (NYTIB). The "reply of the agent" is an EP-1 variable which gets sent to the external system by other rules in the set.

EP-1 currently employs generalization only in limited ways. It generalizes the response of the external system by considering only the first few characters of the last line of output. This was adequate for many tasks, but to really expand the range of tasks it could handle EP-1 would need specific knowledge about the tasks including rules describing which components of the system response are likely to be invariant. EP-1 also generalizes the task agents by permitting the user to specify which replies to the external system are invariant and which should be acquired by the task agent as it is executing. The result is an interactive task agent that queries the user and passes his replies to the system. In the above example the user might indicate that the date range and the boolean expression defining the range of the search were variables. In this case the task agent would query the user for these values instead of always retrieving abstracts concerning the PLO from April to June, 1976. See Waterman (1978b) for more detail.
DATA ACQUISITION

The term data acquisition will be used here to refer to acquiring knowledge in declarative form. In RITA data acquisition consists of augmenting the RITA data base, i.e., creating or modifying the RITA object-attribute-value triples. Although data acquisition is much easier than program creation, user agents which employ it may exhibit many of the dynamic characteristics of the more powerful program-creating agents. The trick is to organize the task agent (the agent being created) so the static, unchanging portion of the knowledge it contains is procedural, in this case, represented as rules. The dynamic, context-dependent portion is then represented as data to direct the execution of the rules. This was the approach taken in the design of both the TEACH agent and the MESSAGE agent. 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.

Tutor and Teach Agents

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 exemplary programming, the first being the use of EP-1 to create agents from examples of programming tasks. The main difference is that here the end result is declarative knowledge in the form of RITA objects, while in the former case the end result was procedural knowledge in the form of rules.

The high-level goal is to create a CAI-type of program capable of teaching users how to access and operate external computer systems or application programs (Wilcox, et al., 1976). The emphasis, however, is on techniques for automatic program creation rather than the development of programming languages for CAI applications. The CAI domain was picked simply to demonstrate the versatility of the exemplary programming methodology, and no attempt is made to expand and develop CAI techniques.

Figure 7 is a protocol of an actual user-TUTOR interaction, in which the user is a LISP expert demonstrating the use of the language. User input is shown in italics, TUTOR agent response in normal type, and system response in boldface type.

TUTOR uses the information gained from the protocol of Figure 7 to create a data base that is appended to 14 short RITA rules which form the domain-independent portion of the TEACH agent. By supplying different data bases these rules can be used to teach a variety of languages or operating systems. TEACH works by acting as an invisible interface between the user and the system the user is trying to learn, passing all standard
% User Tutor

_tutor_
What is the name of the system to be demonstrated?
_lisp_
What prompt character does lisp use?

% sys
Please type a command to call the system to be demonstrated
Type text that would be appropriate as a response to a query for help in the lisp system. Terminate the text with:

% carry return
Type "show me" followed by "functions" or a name of a function to elic it 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.

terminate 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)(a d)))
(a b)
-> (end of example)
-> (end of demo)

What are you going to demonstrate? _setq_
-> (setq a 13)
12
-> a
12
-> (plus a 13)
25
-> (setq b '((a b e f))
(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.
%
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 by either 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 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, and "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: INTRO'S, TEXT'S, and DEMO'S. 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 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 the TEACH agent the INTRO object with the name "what?" is automatically accessed and its text displayed. A protocol of an actual user-TEACH interaction is shown in Figure 8. The user is interacting with the TEACH agent that was created from the user-TUTOR dialogue of Figure 7. User input is shown in italics, TEACH agent response in normal type, and system response in boldface type.

The TEACH agent 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 ARPA, 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, such as the user agents created by EP-1.
% 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 setq
I don't know about setq

-> show me setq
[start of setq demo]
-> (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)
fist
-> (setq b '1(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)
% exit;
exiting.
%

Figure 8. A Protocol of the Use of the Teach Agent Created for Lisp
Writer and Message Agents

WRITER is an active agent that extracts from the user the information it needs to help him create a reactive message, i.e., an interactive message agent that can be sent to a recipient, in much the same fashion as computer mail is now delivered. WRITER is similar to TUTOR in that both are dynamic agents that map information obtained from the user into declarative knowledge in the form of RITA objects. MESSAGE, the task agent created by WRITER, is similar to TEACH since it also consists of a small number of domain-independent RITA rules which are driven by a large data base. In this case the data provides a specification of the dialogue desired between the reactive message and the recipient.

The reactive message is an offshoot of an earlier concept, the interactive letter (Anderson and Gillogly, 1976; 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 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. Examples include form letters, questionnaires, time sheets, etc. Reactive messages 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 (see Waterman 1978a for more detail on reactive messages).

There are a number of problems involved in implementing a message-writing agent. The main problem is to allow the message to take the form of a general net structure and still have it 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 a simplified type of net is permitted, but it is clear that simple ways of specifying general interconnections in the net are needed.

MESSAGE, the reactive message implemented in RITA, is 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.  

Figure 9 shows a typical MESSAGE data base created by a user-WRITER interaction. Note that there are two types of nodes in the net: statement nodes and question nodes. Similarly there
"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."

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

Yes

"Good"

No

"Why not?"

*  

"What time, between 11 a.m. and 4 p.m., is best for you?"

"OK, thanks anyway for your response."

12

1

*  

"Noon is a bad time for most people."

"That sounds good to me."

"I see."

*  

"What's your second choice?"

*  

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

Figure 9. Data base for a reactive message
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 user-agent interaction this message could 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.

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 elicits 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.

Care must be taken during reactive message
construction to keep the user from being forced down a path he doesn't want to take. The saving factor here is that at each point in the dialogue the user's entire response is saved by the agent for transmittal back to the sender. Thus when the user cannot find an alternative he likes, he can pick one anyway, qualifying his reply with extra comments.

As currently designed, the reactive message is a small, unsophisticated program for dialogue elicitation and could be extended a number of ways to make it a more effective communication tool. One extension is to organize the data base as a complex net rather than a tree. Then interconnections between arbitrary nodes eliminate redundancy, reducing the size of the data base so it won't grow exponentially as questions are added. In the current implementation arbitrary nets are not permitted but some interconnection of nodes is allowed. A second extension involves back-up. Since the recipient of a reactive message can't look ahead to see what's coming he may make mistakes or change his mind regarding previous answers. He then needs the capability for backing up to a previous answer and starting fresh from that point. He may also want to start over from the beginning, or even abort the message so that nothing is returned to the sender. A third extension would be to return machine-readable output to the sender. Besides transmitting the complete protocol of the user-agent interaction a mapping of the user replies onto the expected replies could 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.
IV. CONCLUSIONS

The work in knowledge acquisition using RITA agents as interface programs has shown that the concept of the user agent is both useful and viable, that a rule-based representation facilitates procedural knowledge acquisition, and that placing all the domain-dependent knowledge in a data base that directs the execution of a relatively small rule set is a feasible way to organize some knowledge-acquisition programs.

The task agents that have been developed, such as those for file transfer and NYTIB retrieval, have proved quite useful since they free the user from the mundane, repetitive tasks involved in periodically accessing external computer systems. The teaching agents offer the advantage of on-line access as compared with typical printed reference manuals. They also provide live, rather than canned, demonstrations, which means that when the system changes, the changes will automatically be reflected in the demonstrations. 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. The reactive message agents have some advantages over conventional forms of computer communication. The act of reading the message can automatically generate a reply, which means that the sender gets "instant response," and the recipient doesn't have to worry
about organizing and forwarding the reply. Also, the reactive message maps the recipient's replies onto a small set of expected responses which could be processed by a second agent that would transmit the end product to the sender (e.g., tabulate the results of all replies to a questionnaire).

There are also certain inherent disadvantages in the use of reactive messages. The sender may find the message difficult to organize and create. Since it is a program, some thought must 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 clearly more work is involved than in creating an ordinary piece of computer mail. The reactive message also presents some special problems to the recipient. For example, he might like to read the message without responding, just skim it to decide whether or not to respond, or during interaction back up to some previous answer and start over. Such capabilities are necessary but would result in a more complex control structure for the reactive message agent.

Although both the dynamic agents and the task agents were organized as production systems, this representation was critical only for the task agents, i.e., the programs being created. The production system modularity paved the way for the needed
incremental creation of the given task agents.
REFERENCES


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


FOOTNOTES


2. The training mode is not currently implemented.

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

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