PPS: A Parsimonious Production System

Arie Covrigaru and David E. Kieras

University of Michigan

Technical Report No. 26 (TR-87/ONR-26)

April 22, 1987

This research was supported by the Office of Naval Research, Personnel and Training Research Programs, under Contract Number N00014-85-K-0138, Contract Authority Identification Number NR 667-543. Reproduction in whole or part is permitted for any purpose of the United States Government.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release: distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-87/ONR-26	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Michigan; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Cognitive Science Office of Naval Research (Code 1142CS) 800 N. Quincy Street
6c. ADDRESS (City, State, and ZIP Code) Technical Communication Program Ann Arbor, MI 48109-1109	7b. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K-0138
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61153N	RR04206	RR04206-0A	NR667-543

11. TITLE (Include Security Classification) PPS: A Parsimonious Production System

12. PERSONAL AUTHOR(S) Arie Covrigaru and David Kieras

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	April 22, 1987	34

16. SUPPLEMENTARY NOTATION Address correspondence to David E. Kieras

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Cognitive models, production systems, rule-based systems
05	10		

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

Production systems are commonly used in cognitive modelling research. However, most production system implementations are difficult to program because of the syntax complexity of the rules, and the presence of obscure interactions between rules resulting from system features such as conflict resolution. PPS is a production system implementation in which simplicity of syntax and explicitness of control structure have been emphasized. Experience with several modelling projects suggests that this approach is well suited for cognitive modelling. This paper describes the production rule syntax, and summarizes the data structures and algorithms used in the implementation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL Susan Chipman	22b. TELEPHONE (Include Area Code) (202) 696-4318; 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Abstract

Production systems are commonly used in cognitive modelling research. However, most production system implementations are difficult to program because of the syntax complexity of the rules, and the presence of obscure interactions between rules resulting from system features such as conflict resolution. PPS is a production system implementation in which simplicity of syntax and explicitness of control structure have been emphasized. Experience with several modelling projects suggests that this approach is well suited for cognitive modelling. This paper describes the production rule syntax, and summarizes the data structures and algorithms used in the implementation.

PPS: A Parsimonious Production System

Arie Covrigaru and David Kieras

Abstract

Production systems are commonly used in cognitive modelling research. However, most production system implementations are difficult to program because of the syntax complexity of the rules, and the presence of obscure interactions between rules resulting from system features such as conflict resolution. PPS is a production system implementation in which simplicity of syntax and explicitness of control structure have been emphasized. Experience with several modelling projects suggests that this approach is well suited for cognitive modelling. This paper describes the production rule syntax, and summarizes the data structures and algorithms used in the implementation.

1. Introduction

The production system form of cognitive architecture is important not only because it has a long history in cognitive modelling and artificial intelligence, but also because it is likely to continue to be popular in the future. Production systems are especially useful in the cognitive modelling domain because they are well suited for the representation of procedural knowledge, and analyses based on production rule representations appear to have enough empirical content to be useful in modelling human behavior in complex tasks (e.g., Anderson, 1983; Kieras & Bovair, 1986; Polson & Kieras, 1985; Thibadeau, Just, & Carpenter, 1982).

The major virtue claimed for this architecture is the simplicity and modularity of the knowledge representation, along with the potential for parallelism. In practice, however, most production system implementations are complex and difficult to program in. This seems to be due to the following two reasons:

- The production rule syntax can be complex, and can reflect commitments to specifics of the cognitive architecture, such as the number and organization of the memory systems. The programming can become unduly difficult if the programmer wants to implement a set of cognitive assumptions different from the committed architecture, or wants to work with very simple data structures.
- It can be difficult for the programmer to predict or control when a particular production rule will fire, since many implementations involve complex "conflict resolution" and "refractory" conventions. These conventions can simplify the programming at the level of individual rules, but the result is often that rules interact in ways not obvious from the rules themselves. For example, a rule may not fire if

another rule, rating higher in the conflict resolution scheme, also has its conditions met. Alternatively, if a rule has fired before, it may be refractory (not fire again) unless its conditions are matched in a way considered "new," which again may not be apparent from the rule itself. These interactions with other rules, and the past firing history, can completely destroy the desirable modularity of the production rules to the point where using this architecture has few programming advantages over coding directly in LISP.

These two factors are obstacles to the wider use of production system models in cognitive modelling research and practical applications. For this reason, we developed a production system implementation, called Parsimonious Production System (PPS), in which simplicity and explicitness of representation and programming was emphasized, at the possible expense of compactness and power. We also wanted an implementation that was coded directly in LISP for portability reasons, and so were willing to deemphasize speed and capacity.

From the cognitive modeller's point of view, the key features of PPS are as follows:

- The syntax of the rules is as simple as possible, and in practice, PPS rules are easy for both the programmer and non-programmer to read and interpret.
- There are no learning mechanisms in PPS - it is intended to support the development of static production systems, rather than ones that change with experience.
- The system's working memory structure is very simple, and allows the programmer to define the memory storage systems just by usage, rather than having a fixed memory architecture.
- The programmer is encouraged to make the control structure in the cognitive model explicit, rather than relying on implicit mechanisms in the production rule interpreter. Since any number of rules may fire at once, and there are no conflict resolution or refractory mechanisms built into the system, the production rules have to make the desired control structure explicit.

Thus PPS can be viewed as an experiment in simplicity of production rule architecture - can significant and useful systems be built within this simple framework?

2. System Overview

2.1. What is PPS ?

PPS is a production system shell. The system consists of two parts: A compiler and an interpreter. The input to PPS is a set of statements of the form <IF Condition **THEN** Action>, called production rules, and an initial state of the system's working memory. The working memory is a collection of the clauses of the form ($a_1 ... a_n$) where a_i $i=1...n$ are constants (atoms in Lisp notation). The following steps are taken by PPS in order to process its input: First, the compiler builds a data flow network that represents the production rule set, preparing them for the interpreter. Then the interpreter executes the productions by iteratively finding the subset whose conditions match objects in the system's working memory and executing the actions of those production rules. Each match-execute iteration is called a *cycle*.

2.2. Why a data flow net is used

The bottleneck of production system interpreters is performing the matching of the conditions in each cycle. Forgy (1979, 1981, 1982) developed an approach in which the conditions are rearranged into a network in such a way that the matching process considers only the modifications of the state of the system from the previous cycle, and not the whole set of conditions. In addition, the structure of the pattern-matching net enables us to test the parts of the conditions of many production rules at once (implicit parallelism).

2.3. Summary of system components

This section is a high-level description of the system's components. PPS consists of two modules: A compiler, which compiles the patterns in the given set of production rules into a data flow net, and an interpreter, which interprets the production rules using the data flow net.

Compiler: The compiler in PPS takes a set of production rules written in their formal syntax and transforms them into the more efficient form used later in the run time (interpreting) stage of the system. A data flow network is created that stores the information represented in the conditions of individual production rules. First a *discrimination net* is built representing the items in each *pattern* in the conditions. Then the compiler builds the *combining net* which represents the relationships of the patterns in each condition. At the bottom of the data flow network each production rule condition is represented by a single node.

Interpreter: The interpreter executes a set of production rules that are compiled by the compiler into a data flow network. The function of the interpreter is to perform the cycles of recognizing the set of production rules to be *fired* and execute their actions. Those cycles are performed until either a specific StopInterpreter action is executed (as one of the actions of a production rule) or the set of production rules to be fired is empty.

A cycle consists of updating the working memory, propagating the changes down the data flow network to update the list of matching production rules, and finally, executing the actions of these production rules. In the current implementation of PPS there is no conflict resolution. All the production rules that have matching conditions are fired. Furthermore, a production rule will fire on each cycle as long as its condition is matched by working memory elements.

The "Working Memory": An element in the working memory is called a *clause*. A clause is a list (in Lisp notation) of atoms that represent constant values. The working memory for PPS is not a separate data structure; rather the working memory is represented by the state of the pattern nodes in the network which represent the patterns in the rule conditions. Such a node has a status and a set of variable bindings (that can be empty). When a clause is added to the working memory, it matches a pattern, and the status of the node representing that pattern is set to ON. The list of variable bindings (if the pattern has variables) is stored with that node.

2.4. Implementation

PPS is implemented on a Xerox 1108 LISP machine in the Interlisp-D environment. Even though the system takes full advantage of the Interlisp-D environment, the code implementing the algorithms was kept very portable and was actually transported easily to the IBM LISP/VM dialect. The interface with the Interlisp-D system is through a collection of menus that appear in a control window. Selecting items in these menus causes compilation of rules or execution of a compiled set. Various debugging facilities are available, such as display of working memory contents, tracing the execution of the rules, recording a selected subset of the trace, displaying and editing of the rules, and displaying a graphical representation of the data flow net for a set of production rules.

3. The Production Rules

3.1. Overview

The production rules in PPS are the language in which the user of the system specifies the algorithm to be performed by the PPS interpreter. In order to specify any condition in the production rule language, it must be possible to express any well formed formula within the syntax of the conditions. In the rest of this section we will formally define the syntax and semantics of the conditions and actions in the production rules.

3.2. Production Rule Set

A set of production rules is an unordered list of rules in the form:

(ProductionRuleName **IF** (Pattern_1 Pattern_2...Pattern_n) **THEN** (Action_1 Action_2...Action_m))

The list following the **IF** part is the condition. If the working memory contains clauses that match all of the patterns in the condition, the rule fires. The part following the **THEN** is the action list. It consists of a sequence of actions to be performed by the interpreter if the production rule fires.

3.3. Production Rule Condition

The condition of a production rule is a conjunction of patterns (P_1 P_2...P_n) where each pattern P_i, i=1,2...n, has the form: (e_1...e_k). Each element e_l, l=1,2,...k is either a constant, a variable designated by a special prefix (the character "?"), or a wildcard (the string "$$$"). The constant elements in a pattern have no interpretation except as strings. Each pattern can also appear in negation form as (NOT P_i). A negated pattern is matched only if there is no clause in working memory that matches the body of the pattern. If a negated pattern contains a variable, the same variable must appear in a non-negated pattern elsewhere in the condition, in order to ensure that the variable has a defined binding if the production rule fires. In addition, the form (NOT P_1 P_2...P_n) is interpreted as the negation of the conjunction of the patterns P_1 P_2...P_n. Any variables appearing in the patterns are treated the same as variables in single negated patterns.

Any boolean function can be represented with one or more production rules. The AND function is represented by having two or more patterns in the same condition, the OR function is implicitly represented by two separate production rules and the NOT function is represented explicitly in the conditions.

3.4. Production Rule Action

The action of a production rule is a list of actions (A_1 A_2...A_m) which are executed in order if the rule fires. Each A_i, i=1,2...m, in the action-list has the form: (fn a_1....a_k) where fn is one of the known PPS functions (AddClause, DeleteClause or StopInterpreter), or a user-defined function, and a_1...a_k are its arguments (the arguments can be either constants or variables whose domain is the condition of the production rule). Each function is required to return either NIL, a string (in which case the interpreter halts), or a list of the form ((list of clauses to add) (list of clauses to delete)) which specifies clauses to be added and removed from working memory.

The action functions are executed sequentially when the production rule is fired. The whole list of actions in a production is executed for each possible set of bindings of variables for that production. The order of the action execution within the set of production rules to be fired is: All the deletions from the working memory in one cycle are performed before the additions so adding and deleting the same element in the same cycle will always add an element to working memory.

4. The Data flow Net

4.1. Overview

The set of production rules are compiled into a network that is a directed graph, starting with one root node and ending with rule nodes, where each rule node represents one production rule in the system. Conceptually, the data flow net is divided into two parts: a discrimination net and a combining net.

4.1.1. Discrimination net

When a new clause is entered into the working memory or an old clause has to be deleted, it is necessary to identify the patterns that it matches. This is done by a standard discrimination net mechanism (Charniak, Riesbeck & McDermott, 1980). The discrimination net starts at the Root-node, contains item-type nodes which represent the items inside a pattern, and the Pattern-nodes. The following is an example of the discrimination net mechanism.

Example: For example consider the patterns (?Person ISA boy), (Fred ISA ?Something) and (?Person ISA ?Something). The discrimination net representing those patterns is shown below. The clause (Fred ISA boy) will match all three patterns. Matching it against the patterns (?Person ISA boy) and (?Person ISA ?Something) will only take matching four items. Thus, instead of iterating over all the clauses in the conditions trying to find the one that matches, the system branches to the small subset of patterns that have the potential of matching directly, based on one item in the clause.

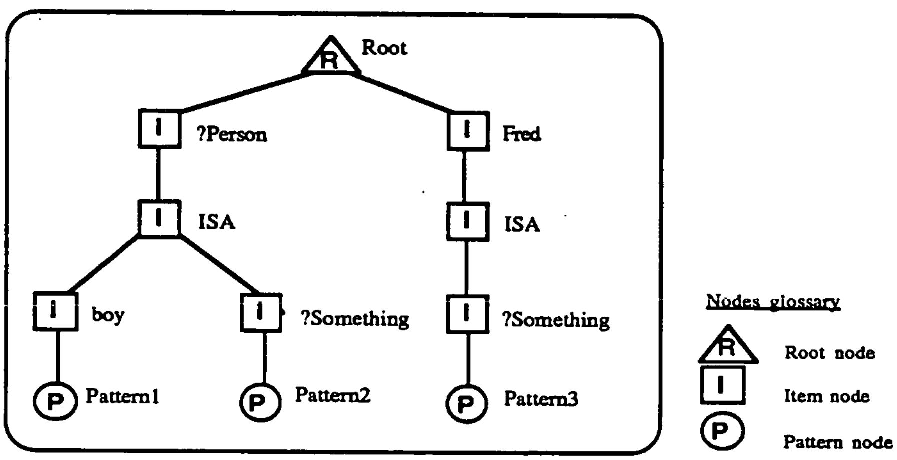

Figure 1: The clause (Fred ISA boy) is matched item-by-item; Fred matches the variable ?Person and the constant Fred, ISA matches at the corresponding constant nodes, and boy matches both the constant boy and the variable ?Something. Thus the three patterns matching this one clause are determined.

4.1.2. Combining net

This part of the net, starting from the nodes representing the patterns, consists of combining nodes that combine the patterns in each rule's condition, keeping account of the structure of the condition and the variable bindings. Each condition is eventually combined into one combining node, and that node points to the rule node, which is a terminal node in the net and represents the corresponding production rule. The combining nodes are either And-node or Negation-node type.

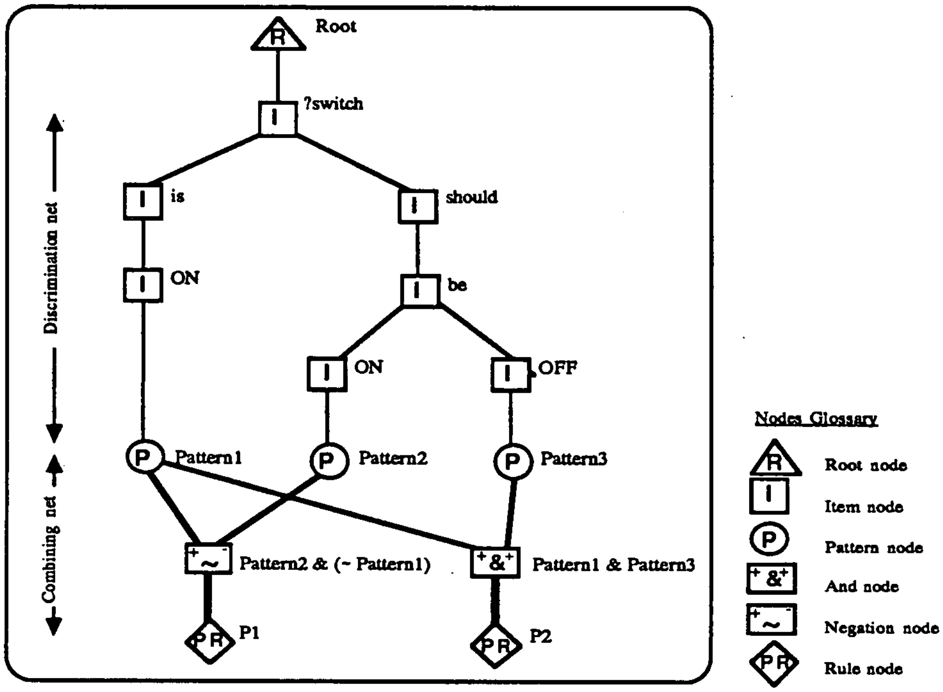

Figure 2: The negation node combines Pattern2 and the negation of Pattern1 into a conjunction that forms the condition of production rule P1; likewise the and node conjoins Pattern1 and Pattern3 to form the condition of production rule P2.

Example: Consider the following two production rules:

```
(P1   IF    (  (NOT(?switch is on))
               (?switch should be on))
      THEN  (  (Turn ?switch on)
               (AddToWorkingMemory ?switch is on)))

(P2   IF    (  (?switch is on)
               (?switch should be off))
      THEN  (  (Turn ?switch off)
               (DeleteFromWorkingMemory ?switch is on))
```

These productions will be compiled into the net in Figure 2. Initially the status of all the nodes is OFF. The labels of the nodes in the discrimination net are the value attributes of each node and the labels in the combining net are the names of the nodes and what they represent in the net

4.2. Node Types, their attributes and their functionality

In this section we will describe the various nodes in the data flow net. This includes their attributes, function, and what they represent according to the original production rules. Besides the specific attributes to each type of node, all the nodes in the net have the type *attribute* and all the non-terminal nodes have a list of their *successors*. Those properties do not appear in the individual node descriptions. The description goes from the top of the net (root node) to the terminal nodes (rule nodes).

4.2.1. Special Nodes

Root-node: This is the root of the net and the only entry point to the net.

AlwaysTrue-node: This node is created in every network specifically for the cases where the condition of a production rule contains only patterns in negation form. The (AlwaysTrue) pattern represented by the AlwaysTrueNode is always in the working memory so a negated node will be paired with it and combined into a negation node in the system.

4.2.2. Intra Pattern nodes

The following are the nodes that represent elements in the patterns, and appears only in the discrimination net.

Constant-node: This is a node representing an constant item in a pattern. A constant is any item that is not a variable or a wildcard. The attribute of this type of node are the value which is the constant the node represents.

Variable-node: This node represents a variable in a pattern. A variable in PPS notation is any string in a pattern that starts with the prefix "?". Its attributes are the value and the name of the variable it represents.

Wildcard-node: This node is used in the net to enable the PPS user to have items in patterns that have the function of matching any item in a clause that is in that position without keeping track of its value. A wildcard in PPS notation is the string "$$$".

Pattern-node: The pattern node type is used to represent the individual patterns in the conditions in production rule set. This node has the attributes consisting of the pattern in its original form (a debugging aid), the list of variable names in the pattern (used by the interpreter for binding), a flag stating if the pattern contains any wildcards, and a status attribute. This last attribute is ON if there is any clause that entered in the working memory that matched this pattern and OFF otherwise. Initially, the status of all pattern nodes is OFF.

4.2.3. Combining nodes

In order to understand the function of the combining nodes described below we need to define *binding sets*. A binding set is a list of the form (varname$_1$ value$_1$ varname$_2$ value$_2$...varname$_k$ value$_k$) where each value$_i$ is the binding of the variable with the name varname$_i$. Each clause entered into the net that matches a pattern with variables will create a set of bindings for that pattern where each variable in the pattern is bound to the corresponding item in the clause.

And node: An and-node has exactly two predecessors which can be any combination of and-nodes, negation-nodes, or pattern-nodes. It represents a combining function analogous to a logical AND of the two predecessors. The attributes of an and-node are the left and right predecessors, the status, the set union and the set intersection of the lists of names of variables from both predecessors and the list of consistent bindings (described below).

If no variables are involved (the predecessors do not have variables) the and node will have its status ON only when both predecessors have their status ON. If either of the predecessors has variables, then the node will be ON only if both predecessors are ON and there are bindings of variables from the predecessors that are *consistent*, meaning that variables that appear for both predecessors have the same bindings.

For example the binding set (?x frog ?y green) is consistent with the binding set (?x frog ?z jumps) because the variable ?x that appears in both sets has the same binding. These binding-sets will be combined into the set (?x frog ?y green ?z jumps). On the other hand the set (?x frog ?y green) is not consistent with the set (?x watermelon ?z green) because the common variable ?x is bound to frog in the first set and to watermelon in the second.

Negation node: This type of node represents a combining function analogous to the logical expression (A & ~B) where A and B are the two predecessors. The positive predecessor A, can be any combining node or pattern node and the negative predecessor B, must be a pattern node. The negation nodes are needed when there is a negated pattern in a condition. This node has the attributes of status, the union of variable names and their intersection from its predecessors, and a list of variable bindings.

If no variables are involved (in the negated predecessor pattern) the negation node will have its status ON only when no clause matching this pattern is in the working memory, and the positive predecessor is ON. If the pattern has variables, then the node will be ON only if there are bindings of variables on the positive node that are *not* consistent with those of the negated pattern's bindings. Thus the negation node takes the positive predecessor's binding sets, and removes any binding sets that are consistent with the negative predecessor's binding sets and passes this smaller set of bindings down.

To illustrate the function of the negation node, suppose the positive predecessor has the binding sets (?animal bear ?color black) and (?animal bear ?color white) and the negative predecessor has the binding set (?animal bear ?color black ?description furry). The binding set (?animal bear ?color black) is consistent with (?animal bear ?color black ?description furry) and so will not be passed down, but the binding set (?animal bear ?color white) will be passed down the net because ?color has different bindings in the positive and negative predecessors.

4.2.4. Terminal Node

Rule-node: This is the terminal node in the network. Its function is to keep track of the information needed to fire a production rule. Its attributes are status (ON if the production rule is among the set to be fired, OFF otherwise), its predecessor, the actions to be performed when the production rule is fired and the list of all of the binding sets passed down from the combining nodes. On firing, the actions are executed once for each set of bindings.

5. The Compiler

The compiler first builds a discrimination net of the different patterns in the conditions, and then constructs a combining net that combines the patterns to represent the condition of each production rule terminating with a single combining node that is a unique predecessor of a rule node.

Note that in PPS, the order in which rules appear in the list of productions, or the order of the patterns in conditions, does not determine which rules will fire, nor the order in which they fire. However, in constructing the combining net, the compiler uses a fast heuristic that takes advantage of the fact that the cognitive model programmer will tend to write the production rules in a certain order for reasons of legibility and clarity. Rules that have condition patterns that appear in many other rules, such as statements of general goals, tend to appear earlier in the list of rules than ones that have condition patterns such as specific goals that appear only in a few, or individual, rules. Likewise, within a rule, more general (frequently occurring) patterns tend to appear first in the rule condition, with more specific tests appearing later. PPS does not require rules to have this ordering property, but it is a natural way to write the rules in a cognitive model.

The compiler can exploit this ordering property in constructing the combining net by simply collecting pairs of patterns in the order that they appear in the rules. The resulting network may not be the optimal one, either in size or run-time speed, but the compilation time with this approach is considerably better than that of an optimizing compiler algorithm we have experimented with, and the run time appears to be close to what an optimizing compiler would produce. Since in a cognitive modelling domain there are many revisions to the model, but only a few "production runs," this tradeoff is the appropriate one.

The following sections summarize the algorithms used by the compiler in the discrimination net construction and the combining phase. For a more detailed description of the compilation algorithms see Appendix A.

5.1. Discrimination Net Phase

This is the procedure of compiling the patterns into a discrimination network to create one node to represent each distinct pattern in a production rule set. Each pattern in the conditions is then replaced by the name of its representative node in the net.

The discrimination algorithm works as follows: It picks up a pattern and, starting at the root node and with the first item in the pattern, it looks for the node representing that item among the immediate successors. If the node is found, the process is repeated with the second item in the pattern and the successors of the representing node. As long as the nodes are found in the net, nothing new is created. If an item has no representing node among the immediate successors, a new node is created to represent the new item and added to the set of successors of the last node. When the compiler exhausts all the items in the pattern it looks for a pattern node among the successors of the last node. If one is found it means that the pattern was encountered in a previous production rule, otherwise a new pattern node is created and added to the set of successors of the node representing the last item. In either case the pattern node (found or newly created) replaces the original pattern in the rule condition. The discrimination procedure is repeated for each pattern in the condition and for each condition in the set of production rules.

Patterns in negation form are treated the same way by the discrimination procedure, but the pattern in the condition is replaced with the name of the pattern node prefixed by the character "~" signifying the fact that the pattern is in negation form. This prefix will be noticed later by the combining net building procedure and used to create negation nodes.

As an example consider the following production rule:

```
(P1    IF    ( (?switch is off)
               (?switch should be on))
```

```
        THEN (  (Turn ?switch on)
                (DeleteFromWorkingMemory ?switch off)
                (AddToWorkingMemory ?switch is on)))
```

Suppose the patterns are represented by the nodes pattern1 and pattern2 (see Figure 2); after the discrimination process it will look like this:

```
(P1     IF      ( pattern1
                  pattern2)
        THEN (  (Turn ?switch on)
                (DeleteFromWorkingMemory ?switch off)
                (AddToWorkingMemory ?switch is on)))
```

5.2. Combining Phase

The second phase in the compilation procedure is to combine the patterns in each condition such that one combining node will represent the condition of each production rule. In general, the compiler builds a net starting from the pattern nodes down to rule nodes. In the course of generating this net, it replaces pairs of nodes in the conditions with combining nodes until each condition consists of only one node. Finally the compiler creates a rule node in the net that holds the information about the action of the production rule and its variable binding sets.

At the beginning of this phase, the conditions are represented as a list of pattern node names. The compilation of a condition is as follows: As long as the condition consists of more than one node name do: Pick the first pair of nodes and determine if they can be combined by either an and node or a negation node. If not, move one of them to the end of the condition and pair the other with the next node name in the condition; If so, search the intersection of those node's successor lists to determine if the combining node was created in a previous compiled condition. If such a node is found, replace the pair in the condition with the name of the combining node and repeat the procedure on the next pair. If a combining node does not exist, create one and replace the pair in the condition with its name. When the condition consists of only one node name, create the terminal rule node, and go on to the next condition.

6. The Interpreter

The interpreter runs in a cycle of matching the conditions and executing the actions of the production rules whose conditions match. This cycle is repeated until either there are no productions that match, or one of the actions stops the system deliberately. In order to give a clear description of the matching algorithm, it is useful to define the "state of the system".

6.1. The "State of the System"

Each cycle of the interpreter can be looked at as a time pulse such that t_1 will be the beginning of the first cycle, t_2 the second and so on.

At time t_i the state of the system is as follows:

- $Database_i$: The contents of the database consist of the set of pairs (status, binding sets) of all of the pattern, combining, and rule nodes in the data flow net.
- $WorkingMemory_i$: A subset of $Database_i$ which is all the pairs (status, binding sets) of pattern nodes only.
- $FiredList_i$: The list of production rules whose conditions match the current contents of the working memory.
- $AddList_i$: The list of clauses to be added to working memory.
- $DeleteList_i$: The list of clauses to deleted from working memory.

Each of the components of the state of the system can be empty at any time, but if FiredList is empty, the interpreter will stop cycling.

6.2. The Matching Algorithm

This section is a summary of the matching algorithm. A more detailed description of the procedures used in the interpreter appears in Appendix B.

The matching process is executed in two conceptually different phases. Phase one is discriminating a clause in the working memory by updating the pattern nodes, and phase two is propagating the changes into the combining net to update the whole database. The second phase results in the updated state of FiredList. In the current implementation it was decided to make the updating changes depth first for each clause in DeleteList and AddList . This approach saves bookkeeping during the propagation of changes and is more straightforward conceptually.

Thus, for each clause in $DeleteList_i$ and $AddList_i$ the interpreter does the following: First it discriminates the clause, finding all the pattern nodes that match the clause. Each pattern node is updated to show whether the clause was added or deleted, and changes to the bindings sets if variables are involved. For each of those nodes the interpreter calculates whether the state of the node has changed, and if so, the changes are propagated to the successors. The update-propagate procedure is repeated for each successor. This depth-first update-propagate process is stopped either when the changes do not change the state of a node, or a rule-node is reached. If the new status of the rule node is ON, the rule is added to FiredList, otherwise the rule is removed from FiredList.

When each clause in $DeleteList_i$ and $AddList_i$ has been processed, the changes in the net results in $Database_{i+1}$ and $FiredList_{i+1}$. Next the actions in the production rules in $FiredList_{i+1}$ are executed, resulting in $DeleteList_{i+1}$ and $AddList_{i+1}$. Then cycle i+1 starts.

6.3. An Example

This example will illustrate how the matching algorithm is superior to a simple production system interpreter in terms of the steps needed to match a simple production rule condition that has two patterns and two variables. The example will be displayed for both, a simple interpreter and the PPS interpreter. For more detailed discussion see Forgy (1982).

Suppose we have the following production in the system:

```
(P1   IF     (  (?switch is ON)
                (?switch is connected to ?light)
                (NOT (?light blinks)))
      THEN   (  (Turn ?switch ON)
                (AddToWorkingMemory ?light blinks)))
```

and the working memory consists of the following items:

(S1 is ON)	(S1 is connected to L56)
(S2 is connected to L3)	(S3 is ON)
(S3 is connected to L1)	(S4 is ON)

A simple interpreter, in order to find out if P1's condition matches the facts in the working memory, would have to find and mark all the clauses that match the first pattern, which is one scan of the working memory, then for each marked clause, find all the clauses that match the second pattern and are consistent with the first. In this example, three more passes over the working memory would be required. If there are more than two patterns the process would grow exponentially. Furthermore, the whole process will be repeated every cycle and for every rule, even if the changes in the working memory are unrelated to the rule.

The same example is treated by the PPS interpreter in a completely different way. First, P1 is compiled into a net as shown in Figure 3. Since the PPS matching algorithm is concerned only with changes in the state of the system, suppose P1 is not in FiredList and all the clauses shown above as facts in working memory are in AddList at this moment.

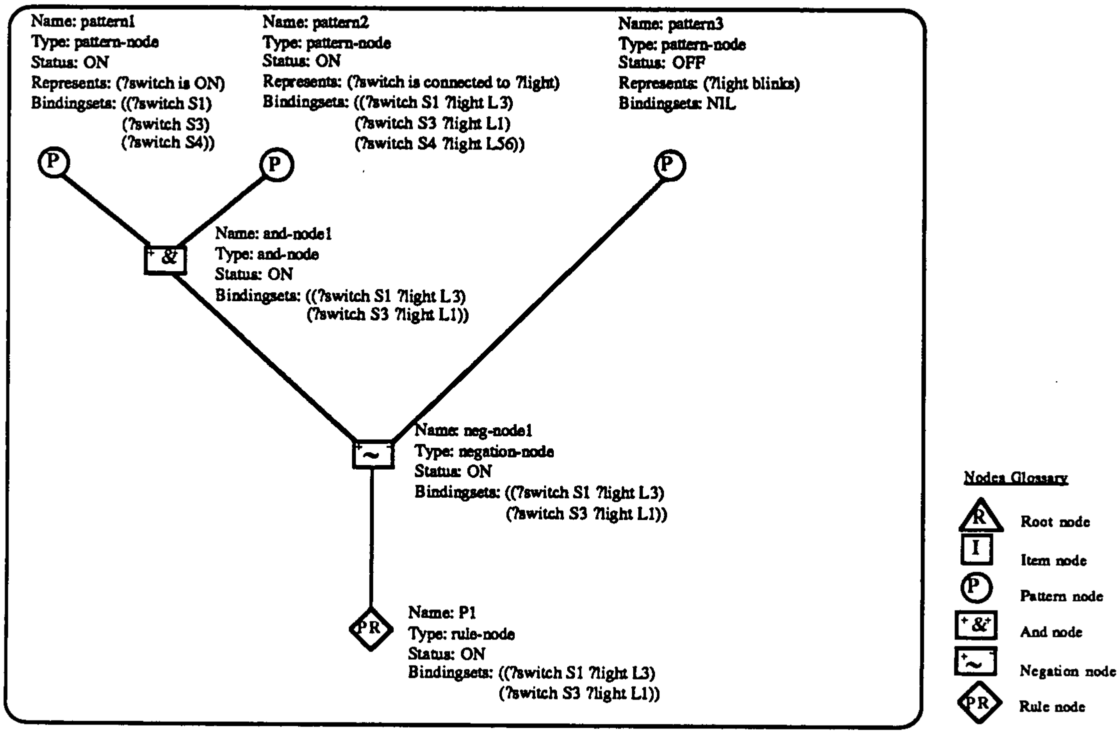

Figure 3: The part of the data flow net for P1 including only the nodes from the patterns down to the rule.

The first step is to update the working memory which takes a time proportional to the number of the different patterns that each clause matches (in this case, just the number of the clauses). Second, the changes are propagated down the net. Updating and-node1 consists of computing the consistent binding sets that were added, which will take matching nine values. Then, one more step is done to set the status of P1 and its bindings. As long as none of the patterns are changed, P1 will be fired each cycle without recomputing its binding sets. This state is shown in Figure 3. If in the next cycle the clauses (L1 blinks) and (L3 blinks) will be added to the working memory, it will change the state of the pattern node pattern3 creating the binding sets (?light L1) and (?light L3) and thus changing the node's status to ON. This fact will require updating the successor of pattern3, in this case removing from the binding sets of neg-node1 all the sets that contain the bindings of ?light to L1 and L3. This will leave neg-node1 with no bindings and will change its status to OFF. Propagating this change again to the successor will turn the status of the rule node P1 to OFF, removing it from FiredList.

7. Conclusions

The PPS package has been used over the last few years to construct several fairly complex models of routine cognitive skill in human-computer interaction (e.g. Polson & Kieras, 1985; Bovair, Kieras, & Polson, in preparation), and a set of models concerning problem-solving with a mental model for a piece of equipment (Kieras, in press; in preparation), and a few other smaller projects. The flexibility of PPS with regard to memory organization, the simple rule syntax, and the explicit representation of the control structure was critical to the easy construction and empirical value of these models. The fact that PPS can be used routinely to easily construct cognitive models shows that the package meets its intended goals fairly well.

However, it seems clear that some of the simplicities of PPS are likely to present serious problems if an attempt is made to extend it to learning situations; for example, learning mechanisms such as Anderson's (1983) are based on the presence of conflict resolution rules that enable newly acquired production rules to eventually dominate previously learned rules. The PPS architecture would have to be changed to implement such learning mechanisms.

Appendix A: Compiler Functions

The following sections describe the algorithms to compile a set of rules. In the compilation process as well as in the matching process, only the conditions of production rules are relevant; the actions and their format plays no role.

Before starting the compilation, the compiler scans the given set of production rules, checking for syntax errors, variable bindings (variables must appear in at least one pattern in positive form in the condition of a production rule), and also translates conditions with negations of conjoined patterns into conditions with simple negations that the compiler understands.

Representing negation of conjunctions is done by replacing the single production rule with several production rules that are logically equivalent to the original rule based on the equivalence of (NOT (A & B & B)) to ((NOT A) v (NOT B) v (NOT C)). In PPS the logical OR is implicit in that two or more production rules are interpreted as a disjunction; in the above example, we can replace one production with three others, each having the condition of one of the disjunctives in the equivalent formula, all with the same action list as the original. This is done transparently to the user.

Discrimination Net Construction

The discrimination procedure is performed by iterating over the list of rules and replacing each pattern in the conditions with the corresponding node name as the net is created. The following algorithm describes the function **Discriminate-Pattern** that is applied to each pattern in the conditions. In this algorithm we assume the definitions the predicates:

(**MatchItem** *item node*) - Matches *item* against *node*. If *node* has the same type as *item* and the value of node is *item* it returns *node*, otherwise, NIL.

(**CreateDiscriminationNode** *item*) - Takes *item*, an element of a pattern, and creates a node in the net, returning the name of the new created node.

(**CreatePatternNode** *node*) - Takes *node*, the last element of a pattern, and creates a node in the net of type pattern-node, returning the name of the new created node.

Procedure: **Discriminate-Pattern**
Input: Root-node, Pattern
Output: Name of pattern node

{Begin Discriminate-Pattern}

1. Set current-node to be the Root-node and successors to be the list of successors of current-node.
2. If there are no more items in the pattern, go to 4; otherwise, set current-item to be the first item in the pattern and remove it from the pattern.
   2.1. Set type to be the type of current-item (constant, variable or wildcard). If the type of current-item is *variable*, add it to the *variable-list*. If the type is wildcard, set wildcard flag to T.
3. For each successor apply **MatchItem** to current-item and the successor. If current-item matches any successor, set that successor to be current-node and go to 2. Otherwise apply **CreateDiscriminationNode** to create a new node to represent current-item, add that node to the successors of current-node. Set current-node to be this new node and go to 2.
4. If among the successors there exists a node of type pattern-node, exit returning that node, otherwise call **CreatePatternNode** to create a new node of type pattern-node to represent the pattern, put the variable list and the wildcard flag on the node, add it to the successors of current node and exit returning the new node.

{End Discriminate-Pattern}

Pattern Combining Net Construction

The combining net construction is performed by iterating over the condition of a rule and replacing each pair of patterns in the conditions with a combining node name as the net is created. The following algorithm describes the function **Compile-condition**. In this algorithm we make use of the predicates:

(**NonsenseCombination** *node1 node2*) - Tests if two given nodes can be validly combined by an and or negation node. The rules for invalid combination of two nodes are:

1) Both nodes are patterns in negation form.
2) *node2* is a pattern in negation form, both nodes have variables but the intersection of the sets of variables is empty.
3) *node2* is a pattern in negation form, with variables, but *node1* does not have variables.
4) *node1* is a pattern in negation form, both nodes have variables, but the intersection of the sets of variables is empty.
5) *node1* is a pattern in negation form with variables, but *node2* does not have variables.

(**DetermineNodeType** *node1 node2*) - Determines the type of a combining node for two given nodes. If either node is a pattern in negation form, the combining node will be a negation node; otherwise it will be an and node.

(**FindCombiningNode** *node1 node2*) - Determines if the two given nodes have a combining node among their successors. If such a node is found, its name is returned; otherwise NIL is returned.

Procedure: **Compile-Condition**
Input: Root-node, Condition
Output: Name of rule node

{Begin Compile-Condition}

1. Set *left-of-pair* to be the first element (node) of the condition and remove it from the condition.
2. If condition is empty go to 7; otherwise, set *right-of-pair* to be the first element and remove it from the condition.
3. Apply **NonsenseCombination** to the pair. If a combination of the two nodes is invalid by the rules above, add left-of-pair to the end of the condition, and go to 1; otherwise go to 4.
4. Determine the type of combining node of the pair by applying **DetermineNodeType** to *left-of-pair* and *right-of-pair*. Apply **FindCombiningNode** to *left-of-pair* and *right-of-pair*. If a combining node exists, go to 6; otherwise go to 5.
5. Create a new combining node for the pair and add it to the list of successors of *left-of-pair* and *right-of-pair*.
6. Add the combining node name to the condition. Go to 1.
7. At this point, *left-of-pair* represents all the patterns in the current condition. Create a new rule node as successor to *left-of-pair*, and exit returning the name of the rule node.

{End Compile-Condition}

Appendix B: Interpreter Functions

This algorithm is performed until the interpreter is stopped by an empty set of production rules to be fired or deliberately by the actions. The algorithm to execute one cycle is as follows:

{Begin Interpreter-Cycle}

1. For each clause in *DeleteList* do
   Update-FiredList (*DeleteList ,root-node, list-of-rules-to-be-fired*)
2. For each clause in *AddList* do
   Update-FiredList (*AddList, root-node, list-of-rules-to-be-fired*)
3. For each *production-rule* in *list-of-rules-to-be-fired* do
   Execute-Actions (*production-rule*)
4. If *list-of-rules-to-be-fired* is empty, stop otherwise go to 1.

{End Interpreter-Cycle}

Update Fired List

This is the matching algorithm of PPS. The procedure performed with the AddList is the same as the procedure with DeleteList so the reader should assume that Update-FiredList is called twice each cycle, first with DeleteList and then again with AddList. The algorithms of the procedures **Discriminate-Clause**, **UpdatePatternNode** and **Propagate-Changes** used here are described in detail later.

Procedure: **Update-FiredList**
Input: list-of-clauses, root-node, list-of-rules-to-be-fired
Output: list-of-rules-to-be-fired

{Begin Update-FiredList}

1. Set *current-clause* to be the first clause in *list-of-clauses* and remove it from *list-of-clauses.*
2. Set *set-of-matching-pattern-nodes* to be the set of patterns returned from applying **Discriminate-Clause** to *current-clause.*
3. Set *current-node* to be the first node in *set-of-matching-pattern-nodes* and remove it from *set-of-matching-pattern-nodes.*
4. Apply **UpdatePatternNode** to *current-node.*
5. Apply **Propagate-Changes** to all the successors of *current-node* that were changed.

6. If *set-of-matching-pattern-nodes* is empty go to 7; otherwise go to 3.
7. If *list-of-clauses* is empty go to 8; otherwise go to 1.
8. Return the updated *list-of-rules-to-be-fired.*

{End Update-FiredList}

Update Working Memory

The following is the algorithm for updating the working memory. This algorithm makes use of the predicate:

(**ItemMatchesNode** *Item Node*) - Determines if *item* matches *node*, considering type and value. If *node* is of type constant, *item* matches if it is identical to the value of *node.* If *node* is of type variable, *item* matches and the binding (variable-name *item*) is recorded. If the type of the node is wildcard, *item* always matches.

Procedure: **Discriminate-Clause**

input: root-node, current-clause (clause to be added or deleted), action (add or delete)

Output: Set of pattern-nodes

{Begin Discriminate-Clause}

1. Set *current-node* to be the root-node.
2. Set *current-item* to be the first item of *current-clause* and remove it from *current-clause.*
3. Set *set-of-matching-nodes* to be all the successors of *current-node* that *current-item* matches by applying **ItemMatchesNode** to *current-item* and each successor.
4. Set *successors* to be the union of all the successors of *set-of-matching-nodes.*
5. If *current-clause* is not empty go to 2 otherwise got to 6.
6. Set *set-of-matching-nodes* to be all the successors of type pattern-node. Mark all the changes (add or delete and bindings) on the record of changes of each pattern-node. Return set-of-matching-nodes.

{End Discriminate-Clause}

Propagate Changes

The changes are kept on each node's change record which has three fields: status change, binding sets addition and binding sets deletions. The process of propagating the changes is done depth-first for each pattern node that was marked as changed by the Discriminate-Clause procedure applied to a clause in DeleteList or AddList.

(**UpdatePatternNode** *Node*) - Updates the state of a pattern node. This function is called when a change to the state of this node was recorded by Discriminate-Clause. The state of the node is determined according to the changes passed down by updating the predecessor, and the changes in the state of the current node are passed to its successors. The function returns the changes made in the state of the current node.

(**UpdateAndNode** *Node*) - Updates the state of an and node. The changes are marked as addition or deletion passed down from one of the predecessors. Those can be addition or deletion of bindings or change in the status of the node if it doesn't consist variables. The change in the predecessor's state is calculated in combination with the other predecessor and if different then the current state of the node the state of the node is updated and the change is passed to its successors, otherwise the function returns NIL and the propagation is stopped on this path.

(**UpdateNegationNode** *Node*) - Updates the state of an negation node. The changes are marked as addition or deletion passed down from one of the predecessors. Those can be addition or deletion of bindings or change in the status of the node if it doesn't consist variables. This node's changes can be more complicated then an the changes in an and node since adding bindings to the negated predecessor will cause removing bindings from this node's output which means removing bindings from the successors. The change in the predecessor's state is calculated in combination with the other predecessor and if different then the current state of the node the state of the node is updated and the change is passed to its successors, otherwise the function returns NIL and the propagation is stopped on this path.

(**UpdateRuleNode** *Node*) - This function is called to modify the state of a rule-node. It adds or deletes the name of the production rule from the list of rules to be fired according to the state of the predecessor to this node. The predecessor represents this production rule's condition. The function returns the name of the node as its result.

Procedure: **Propagate-Changes**
Input: Current-node, List-of-rules-to-be-fired
Output: List-of-rules-to-be-fired

{Begin Propagate-Changes}

1. Set *node-type* to be the type of *current-node* (pattern-node, and-node, negation-node or rule-node).
   1.1. If *node-type* is pattern-node set *changes* to be the value returned from **UpdatePatternNode**. Go to 2.
   1.2. If *node-type* is and-node set *changes* to be the value returned from **UpdateAndNode**. Go to 2.
   1.3. If *node-type* is negation-node set *changes* to be the value returned from **UpdateNegationNode**. Go to 2.
   1.4. If *node-type* is rule-node set *changes* to be the value returned from **UpdateRuleNode**. Go to 2.
2. If the state of *current-node* was changed (*changes* is not empty) set *set-of-nodes-to-update* to be the set of all the successors of *current-node*. Apply **Propagate-Changes** to each of the nodes in *set-of-nodes-to-update* and return *current-node*; otherwise stop and return NIL.

{End Propagate-Changes}

References

1. Anderson, John R. (1983). The Architecture of Cognition. Cambridge, Massachusetts: Harvard University Press.

2. Bovair, S., Kieras, D. E., and Polson, P. G. (in preparation). The acquisition and performance of text-editing skill: A production-system analysis. University of Michigan, Ann Arbor.

3. Brownston, L., & Farrell, R., & Kant, E., & Martin, N. (1986). Programming Expert Systems in OPS5. U.S.A.: Addison Wesley.

4. Charniak, E., & Riesbeck, C. K. & McDermott, D. V. (1980). Artificial Intelligence Programming. USA: Lawrence Erlbaum Associates.

5. Forgy, Charles L. (February 1979). On the Efficient Implementation of Production Systems. Doctoral dissertation, Carnegie-Mellon University, Pittsburg, PA, U.S.A.

6. Forgy, Charles L. (July, 1979). OPS4 User's Manual. (Technical Report CMU-CS-79-132). Carnegie-Mellon University: Department of Computer Science, Carnegie-Mellon University.

7. Forgy, Charles L. (July 1981). OPS5 User's Manual. (Technical Report CMU-CS-81-135). Department of Computer Science, Carnegie-Mellon University.

8. Forgy, Charles L. (September 1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. Artificial Intelligence, 19(1), pp. 17-37.

9. Kieras, D. E. (in preparation). Comparison of a production-rule model with engineering models in a device operation task. University of Michigan, Ann Arbor.

10. Kieras, D. E. (in press). The role of cognitive simulation models in the development of advanced training and testing systems. In Frederiksen, N., Glaser, R., Lesgold, A, and Shafto, M. (Eds.), Diagnostic Monitoring of Skill and Knowledge Acquisition. Hillsdale, N.J.: Erlbaum.

11. Kieras, D. E., & Bovair, S. (1986) The acquisition of procedures from text: A production-system analysis of transfer of training. Journal of Memory and Language, 25, pp. 507-524.

12. Kieras, D. E., & Polson, P. G. (1985). Final Report on SUR Project. User Complexity of Devices and Systems.

13. Polson, P. G., & Kieras, D. E. (1985). A quantitative model of the learning and performance of text editing knowledge. CHI'85 Conference Proceedings, (pp. 207-212).

14. Thibadeau, R., Just, M. A., & Carpenter, P. A. (1982). A model of the time course and content of reading. Cognitive Science, 6, pp. 157-203.

1987/02/03 Distribution List
[Michigan/Kieras] NR 667-547

Dr. Beth Adelson
Department of Computer Science
Tufts University
Medford, MA 02155

AFOSR,
Life Sciences Directorate
Bolling Air Force Base
Washington, DC 20332

Dr. Robert Ahlers
Code N711
Human Factors Laboratory
Naval Training Systems Center
Orlando, FL 32813

Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. John Allen
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Dr. William E. Alley
AFHRL/MOT
Brooks AFB, TX 78235

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Technical Director, ARI
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Patricia Baggett
University of Colorado
Department of Psychology
Box 345
Boulder, CO 80309

Dr. Eva L. Baker
UCLA Center for the Study
of Evaluation
145 Moore Hall
University of California
Los Angeles, CA 90024

Dr. James D. Baker
Director of Automation
Allen Corporation of America
401 Wythe Street
Alexandria, VA 22314

Dr. Meryl S. Baker
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. James Ballas
Georgetown University
Department of Psychology
Washington, DC 20057

Dr. Harold Bamford
National Science Foundation
1800 G Street, N.W.
Washington, DC 20550

prof. dott. Bruno G. Bara
Unita di ricerca di
intelligenza artificiale
Universita di Milano
20122 Milano - via F. Sforza 23
ITALY

Dr. Isaac Bejar
Educational Testing Service
Princeton, NJ 08450

Leo Beltracchi
United States Nuclear
Regulatory Commission
Washington DC 20555

Dr. John Black
Teachers College
Columbia University
525 West 121st Street
New York, NY 10027

Dr. Arthur S. Blaiwes
Code N711
Naval Training Systems Center
Orlando, FL 32813

Dr. R. Darrell Bock
University of Chicago
NORC
6030 South Ellis
Chicago, IL 60637

Dr. Deborah A. Boehm-Davis
Department of Psychology
George Mason University
4400 University Drive
Fairfax, VA 22030

Dr. Sue Bogner
Army Research Institute
ATTN: PERI-SF
5001 Eisenhower Avenue
Alexandrai, VA 22333-5600

Dr. Gordon H. Bower
Department of Psychology
Stanford University
Stanford, CA 94306

Dr. Richard Braby
NTSC Code 10
Orlando, FL 32751

Dr. Robert Breaux
Code N-095R
Naval Training Systems Center
Orlando, FL 32813

Commanding Officer
CAPT Lorin W. Brown
NROTC Unit
Illinois Institute of
Technology
3300 S. Federal Street
Chicago, IL 60616-3793

Dr. John S. Brown
XEROX Palo Alto Research
Center
3333 Coyote Road
Palo Alto, CA 94304

Dr. John Bruer
The James S. McDonnell
Foundation
Univ. Club Tower, Suite 1610
1034 South Brentwood Blvd.
St. Louis, MO 63117

Dr. Bruce Buchanan
Computer Science Department
Stanford University
Stanford, CA 94305

Mr. Donald C. Burgy
General Physics Corp.
10650 Hickory Ridge Rd.
Columbia, MD 21044

Maj. Hugh Burns
AFHRL/IDE
Lowry AFB, CO 80230-5000

Dr. Patricia A. Butler
OERI
555 New Jersey Ave., NW
Washington, DC 20208

Joanne Capper
Center for Research into
Practice
1718 Connecticut Ave., N.W.
Washington, DC 20009

Dr. Pat Carpenter
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. John M. Carroll
IBM Watson Research Center
User Interface Institute
P.O. Box 218
Yorktown Heights, NY 10598

Dr. Robert Carroll
OP 01B7
Washington, DC 20370

LCDR Robert Carter
Office of the Chief
of Naval Operations
OP-01B
Pentagon
Washington, DC 20350-2000

Dr. Fred Chang
Strategic Technology Division
Pacific Bell
2600 Camino Ramon
Rm. 3S-453
San Ramon, CA 94583

Dr. Davida Charney
English Department
Penn State University
University Park, PA 16802

Dr. Eugene Charniak
Brown University
Computer Science Department
Providence, RI 02912

Dr. L. J. Chmura
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Dr. Yee-Yeen Chu
Perceptronics, Inc.
21111 Erwin Street
Woodland Hills, CA 91367-3713

Dr. William Clancey
Stanford University
Knowledge Systems Laboratory
701 Welch Road, Bldg. C
Palo Alto, CA 94304

Dr. Charles Clifton
Tobin Hall
Department of Psychology
University of
Massachusetts
Amherst, MA 01003

Dr. Stanley Collyer
Office of Naval Technology
Code 222
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. Lynn A. Cooper
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

LT Judy Crookshanks
Chief of Naval Operations
OP-112G5
Washington, DC 20370-2000

Phil Cunniff
Commanding Officer, Code 7522
Naval Undersea Warfare
Engineering
Keyport, WA 98345

CAPT P. Michael Curran
Office of the CNO
Director, Naval Medicine
Pentagon, Room 4D471, OP-939
Washington, DC 20350-2000

Dr. Cary Czichon
Intelligent Instructional
Systems
Texas Instruments AI Lab
P.O. Box 660245
Dallas, TX 75266

Brian Dallman
3400 TTW/TTGXS
Lowry AFB, CO 80230-5000

Dr. Natalie Dehn
Department of Computer and
Information Science
University of Oregon
Eugene, OR 97403

Goery Delacote
Directeur de L'informatique
Scientifique et Technique
CNRS
15, Quai Anatole France
75700 Paris FRANCE

Dr. Thomas E. DeZern
Project Engineer, AI
General Dynamics
PO Box 748
Fort Worth, TX 76101

Dr. Andrea di Sessa
University of California
School of Education
Tolman Hall
Berkeley, CA 94720

Dr. Stephanie Doan
Code 6021
Naval Air Development Center
Warminster, PA 18974-5000

Dr. Emanuel Donchin
University of Illinois
Department of Psychology
Champaign, IL 61820

Defense Technical
Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC
(12 Copies)

Dr. Jean-Pierre Dupuy
Ecole Polytechnique
Crea 1 Rue Descartes
Paris, FRANCE 75005

Mr. Ralph Dusek
ARD Corporation
5457 Twins Knolls Road
Suite 400
Columbia, MD 21045

Edward E. Eddowes
CNATRA N301
Naval Air Station
Corpus Christi, TX 78419

Dr. William Epstein
University of Wisconsin
W. J. Brogden Psychology Bldg.
1202 W. Johnson Street
Madison, WI 53706

Dr. Edward Esty
Department of Education, OERI
Room 717D
1200 19th St., NW
Washington, DC 20208

Dr. Beatrice J. Farr
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Marshall J. Farr
Farr-Sight Co.
2520 North Vernon Street
Arlington, VA 22207

Dr. Pat Federico
Code 511
NPRDC
San Diego, CA 92152-6800

Dr. Paul Feltovich
Southern Illinois University
School of Medicine
Medical Education Department
P.O. Box 3926
Springfield, IL 62708

Mr. Wallace Feurzeig
Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

Dr. Craig I. Fields
ARPA
1400 Wilson Blvd.
Arlington, VA 22209

Dr. Gerhard Fischer
University of Colorado
Department of Computer Science
Boulder, CO 80309

J. D. Fletcher
9931 Corsica Street
Vienna VA 22180

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Dr. Norman Frederiksen
Educational Testing Service
Princeton, NJ 08541

Dr. Michael Friendly
Psychology Department
York University
Toronto ONT
CANADA M3J 1P3

Dr. Michael Genesereth
Stanford University
Computer Science Department
Stanford, CA 94305

Dr. Herbert Ginsburg
Teachers College
Columbia University
525 West 121st Street
New York, NY 10027

Lee Gladwin
Route 3 -- Box 225
Winchester, VA 22601

Dr. Robert Glaser
Learning Research
& Development Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Dr. Arthur M. Glenberg
University of Wisconsin
W. J. Brogden Psychology Bldg.
1202 W. Johnson Street
Madison, WI 53706

Dr. Marvin D. Glock
13 Stone Hall
Cornell University
Ithaca, NY 14853

Dr. Sam Glucksberg
Department of Psychology
Princeton University
Princeton, NJ 08540

Dr. Daniel Gopher
Industrial Engineering
& Management
TECHNION
Haifa 32000
ISRAEL

Dr. Sherrie Gott
AFHRL/MODJ
Brooks AFB, TX 78235

Dr. James G. Greeno
University of California
Berkeley, CA 94720

Dr. Dik Gregory
Behavioral Sciences Division
Admiralty Research
Establishment
Teddington
Middlesex, ENGLAND

Prof. Edward Haertel
School of Education
Stanford University
Stanford, CA 94305

Dr. Henry M. Halff
Halff Resources, Inc.
4918 33rd Road, North
Arlington, VA 22207

Dr. Ronald K. Hambleton
Prof. of Education & Psychology
University of Massachusetts
at Amherst
Hills House
Amherst, MA 01003

Dr. Wayne Harvey
Center for Learning Technology
Educational Development Center
55 Chapel Street
Newton, MA 02160

Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

Dr. Joan I. Heller
505 Haddon Road
Oakland, CA 94606

Dr. Shelly Heller
Department of Electrical Engi-
neering & Computer Science
George Washington University
Washington, DC 20052

Dr. Per Helmersen
University of Oslo
Department of Psychology
Box 1094
Oslo 3, NORWAY

Dr. John Holland
University of Michigan
2313 East Engineering
Ann Arbor, MI 48109

Dr. Thomas Holzman
Lockheed Georgia
Dept. 64-31
Zone 278
Marietta, GA 30063

Ms. Julia S. Hough
Lawrence Erlbaum Associates
6012 Greene Street
Philadelphia, PA 19144

Dr. James Howard
Dept. of Psychology
Human Performance Laboratory
Catholic University of
America
Washington, DC 20064

Dr. Barbara Hutson
Virginia Tech
Graduate Center
2990 Telestar Ct.
Falls Church, VA 22042

Dr. Alice Isen
Department of Psychology
University of Maryland
Catonsville, MD 21228

Dr. R. J. K. Jacob
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Neil Jacobstein
Manager, Research and
Advanced Development
Teknowledge, Inc.
525 University Ave.
Palo Alto, CA 94301-1982

COL Dennis W. Jarvi
Commander
AFHRL
Brooks AFB, TX 78235-5601

Dr. Robin Jeffries
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971

CDR Tom Jones
ONR Code 125
800 N. Quincy Street
Arlington, VA 22217-5000

Mr. Daniel B. Jones
U.S. Nuclear Regulatory
Commission
Division of Human Factors
Safety
Washington, DC 20555

Dr. Douglas H. Jones
Thatcher Jones Associates
P.O. Box 6640
10 Trafalgar Court
Lawrenceville, NJ 08648

Dr. Jane Jorgensen
University of Oslo
Institute of Psychology
Box 1094, Blindern
Oslo, NORWAY

Dr. Ruth Kanfer
University of Minnesota
Department of Psychology
Elliott Hall
75 E. River Road
Minneapolis, MN 55455

Dr. Milton S. Katz
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Frank Keil
Department of Psychology
Cornell University
Ithaca, NY 14850

Dr. Wendy Kellogg
IBM T. J. Watson Research Ctr.
P.O. Box 218
Yorktown Heights, NY 10598

Dr. Dennis Kibler
University of California
Department of Information
and Computer Science
Irvine, CA 92717

Dr. Peter Kincaid
Training Analysis
& Evaluation Group
Department of the Navy
Orlando, FL 32813

Dr. Walter Kintsch
Department of Psychology
University of Colorado
Campus Box 345
Boulder, CO 80302

Dr. Paula Kirk
Oakridge Associated Universities
University Programs Division
P.O. Box 117
Oakridge, TN 37831-0117

Dr. David Klahr
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Janet L. Kolodner
Georgia Institute of Technology
School of Information
& Computer Science
Atlanta, GA 30332

Dr. David H. Krantz
2 Washington Square Village
Apt. # 15J
New York, NY 10012

Dr. Benjamin Kuipers
University of Texas at Austin
Department of Computer Sciences
T.S. Painter Hall 3.28
Austin, TX 78712

Dr. David R. Lambert
Naval Ocean Systems Center
Code 441T
271 Catalina Boulevard
San Diego, CA 92152-6800

Dr. Pat Langley
University of California
Department of Information
and Computer Science
Irvine, CA 92717

Dr. Marcy Lansman
University of North Carolina
The L. L. Thurstone Lab.
Davie Hall 013A
Chapel Hill, NC 27514

Dr. R. W. Lawler
ARI 6 S 10
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

Dr. Alan M. Lesgold
Learning Research and
Development Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Jim Levin
Department of
Educational Psychology
210 Education Building
1310 South Sixth Street
Champaign, IL 61820-6990

Dr. John Levine
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Michael Levine
Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

Dr. Clayton Lewis
University of Colorado
Department of Computer Science
Campus Box 430
Boulder, CO 80309

Matt Lewis
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Don Lyon
P. O. Box 44
Higley, AZ 85236

Vern Malec
NPRDC, Code P-306
San Diego, CA 92152-6800

Dr. Jane Malin
Mail Code SR 111
NASA Johnson Space Center
Houston, TX 77058

Dr. William L. Maloy
Chief of Naval Education
and Training
Naval Air Station
Pensacola, FL 32508

Dr. Elaine Marsh
Naval Research Laboratory
Code 7510
4555 Overlook Avenue, Southwest
Washington, DC 20375-5000

Dr. Sandra P. Marshall
Dept. of Psychology
San Diego State University
San Diego, CA 92182

Dr. Richard E. Mayer
Department of Psychology
University of California
Santa Barbara, CA 93106

Dr. Gail McKoon
CAS/Psychology
Northwestern University
1859 Sheridan Road
Kresge #230
Evanston, IL 60201

Dr. Joe McLachlan
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. James S. McMichael
Navy Personnel Research
and Development Center
Code 05
San Diego, CA 92152

Dr. Barbara Means
Human Resources
Research Organization
1100 South Washington
Alexandria, VA 22314

Dr. Douglas L. Medin
Department of Psychology
University of Illinois
603 E. Daniel Street
Champaign, IL 61820

Dr. Jose Mestre
Department of Physics
Hasbrouck Laboratory
University of Massachusetts
Amherst, MA 01003

Dr. Al Meyrowitz
Office of Naval Research
Code 1133
800 N. Quincy
Arlington, VA 22217-5000

Dr. Ryszard S. Michalski
University of Illinois
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801

Prof. D. Michie
The Turing Institute
36 North Hanover Street
Glasgow G1 2AD, Scotland
UNITED KINGDOM

Dr. George A. Miller
Department of Psychology
Green Hall
Princeton University
Princeton, NJ 08540

Dr. Lance Miller
IBM-FSD Headquarters
6600 Rockledge Drive
Bethesda, MD 20817

Dr. Andrew R. Molnar
Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

Dr. William Montague
NPRDC Code 13
San Diego, CA 92152-6800

Mr. Melvin D. Montemerlo
NASA Headquarters
RTE-6
Washington, DC 20546

Dr. Nancy Morris
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Randy Mumaw
Program Manager
Training Research Division
HumRRO
1100 S. Washington
Alexandria, VA 22314

Dr. Allen Munro
Behavioral Technology
Laboratories - USC
1845 S. Elena Ave., 4th Floor
Redondo Beach, CA 90277

Dr. David Navon
Institute for Cognitive Science
University of California
La Jolla, CA 92093

Mr. William S. Neale
HQ ATC/TTA
Randolph AFB, TX 78150

Dr. T. Niblett
The Turing Institute
36 North Hanover Street
Glasgow G1 2AD, Scotland
UNITED KINGDOM

Dr. A. F. Norcio
Computer Science and Systems
Code: 7590
Information Technology Division
Naval Research Laboratory
Washington, DC 20375

Commanding Officer,
Naval Research Laboratory
Code 2627
Washington, DC 20390

Dr. Harold F. O'Neil, Jr.
School of Education - WPH 801
Department of Educational
Psychology & Technology
University of Southern
California
Los Angeles, CA 90089-0031

Dr. Michael Oberlin
Naval Training Systems Center
Code 711
Orlando, FL 32813-7100

Dr. James B. Olsen
Director,
Waterford Testing Center
1681 West 820 North
Provo, UT 84601

Office of Naval Research,
Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142BI
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142
800 N. Quincy St.
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142PS
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142CS
800 N. Quincy Street
Arlington, VA 22217-5000
(6 Copies)

Special Assistant for Marine
Corps Matters,
ONR Code 00MC
800 N. Quincy St.
Arlington, VA 22217-5000

Dr. Judith Orasanu
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

CDR R. T. Parlette
Chief of Naval Operations
OP-112G
Washington, DC 20370-2000

Dr. James Paulson
Department of Psychology
Portland State University
P.O. Box 751
Portland, OR 97207

Dr. Douglas Pearse
DCIEM
Box 2000
Downsview, Ontario
CANADA

Dr. Virginia E. Pendergrass
Code 711
Naval Training Systems Center
Orlando, FL 32813-7100

Military Assistant for Training
and
Personnel Technology,
OUSD (R & E)
Room 3D129, The Pentagon
Washington, DC 20301-3080

LCDR Frank C. Petho, MSC, USN
CNATRA Code N36, Bldg. 1
NAS
Corpus Christi, TX 78419

Dr. Steven Pinker
Department of Psychology
E10-018
M.I.T.
Cambridge, MA 02139

Dr. Tjeerd Plomp
Twente University of Technology
Department of Education
P.O. Box 217
7500 AE ENSCHEDE
THE NETHERLANDS

Dr. Peter Polson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Steven E. Poltrock
MCC,
Human Interface Program
3500 West Balcones Center Dr.
Austin, TX 78759

Dr. Mary C. Potter
Department of Psychology
MIT (E-10-032)
Cambridge, MA 02139

Dr. Joseph Psotka
ATTN: PERI-1C
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

Dr. James A. Reggia
University of Maryland
School of Medicine
Department of Neurology
22 South Greene Street
Baltimore, MD 21201

Dr. Wesley Regian
AFHRL/MOD
Brooks AFB, TX 78235

Dr. Gil Ricard
Mail Stop C04-14
Grumman Aerospace Corp.
Bethpage, NY 11714

Mark Richer
1041 Lake Street
San Francisco, CA 94118

William Rizzo
Code 712
Naval Training Systems Center
Orlando, FL 32813

Dr. Linda G. Roberts
Science, Education, and
Transportation Program
Office of Technology Assessment
Congress of the United States
Washington, DC 20510

Dr. Ernst Z. Rothkopf
AT&T Bell Laboratories
Room 2D-456
600 Mountain Avenue
Murray Hill, NJ 07974

Dr. William B. Rouse
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Roger Schank
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

Dr. Janet Schofield
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Karen A. Schriver
Department of English
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Hans-Willi Schroiff
Institut fuer Psychologie
der RWTH Aachen
Jaegerstrasse zwischen 17 u. 19
5100 Aachen
WEST GERMANY

Dr. Judith Segal
OERI
555 New Jersey Ave., NW
Washington, DC 20208

Dr. Robert J. Seidel
US Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

Dr. Colleen M. Seifert
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Ramsay W. Selden
Assessment Center
CCSSO
Suite 379
400 N. Capitol, NW
Washington, DC 20001

Dr. Daniel Sewell
Search Technology, Inc.
5550-A Peachtree Parkway
Technology Park/Summit
Norcross, GA 30092

Dr. Michael G. Shafto
ONR Code 1142CS
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. Sylvia A. S. Shafto
Department of
Computer Science
Towson State University
Towson, MD 21204

Dr. Ben Shneiderman
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Dr. Ted Shortliffe
Computer Science Department
Stanford University
Stanford, CA 94305

Dr. Valerie Shute
AFHRL/MOE
Brooks AFB, TX 78235

Mr. Raymond C. Sidorsky
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Robert S. Siegler
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

LTCOL Robert Simpson
Defense Advanced Research
Projects Administration
1400 Wilson Blvd.
Arlington, VA 22209

Dr. Zita M Simutis
Instructional Technology
Systems Area
ARI
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Derek Sleeman
Dept. of Computing Science
King's College
Old Aberdeen
AB9 2UB
UNITED KINGDOM

Dr. Gail Slemon
Logicon
P.O. Box 85158
San Diego, CA 92138

Dr. Linda B. Smith
Department of Psychology
Indiana University
Bloomington, IN 47405

Dr. Alfred F. Smode
Senior Scientist
Code 07A
Naval Training Systems Center
Orlando, FL 32813

Dr. Richard E. Snow
Department of Psychology
Stanford University
Stanford, CA 94306

Dr. Elliot Soloway
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Paul Speckman
University of Missouri
Department of Statistics
Columbia, MO 65201

Dr. Kathryn T. Spoehr
Brown University
Department of Psychology
Providence, RI 02912

Dr. Marian Stearns
SRI International
333 Ravenswood Ave.
Room B-S324
Menlo Park, CA 94025

Dr. Frederick Steinheiser
CIA-ORD
612 Ames
Washington, DC 20505

Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

Dr. David Stone
KAJ Software, Inc.
3420 East Shea Blvd.
Suite 161
Phoenix, AZ 85028

Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

Dr. Kikumi Tatsuoka
CERL
252 Engineering Research
Laboratory
Urbana, IL 61801

Dr. Martin M. Taylor
DCIEM
Box 2000
Downsview, Ontario
CANADA

Dr. Perry W. Thorndyke
FMC Corporation
Central Engineering Labs
1185 Coleman Avenue, Box 580
Santa Clara, CA 95052

Major Jack Thorpe
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Dr. Sharon Tkacz
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Martin A. Tolcott
3001 Veazey Terr., N.W.
Apt. 1617
Washington, DC 20008

Dr. Douglas Towne
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

Dr. Kurt Van Lehn
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. Jerry Vogt
Navy Personnel R&D Center
Code 51
San Diego, CA 92152-6800

Dr. Ming-Mei Wang
Lindquist Center
for Measurement
University of Iowa
Iowa City, IA 52242

Roger Weissinger-Baylon
Department of Administrative
Sciences
Naval Postgraduate School
Monterey, CA 93940

Dr. Donald Weitzman
MITRE
1820 Dolley Madison Blvd.
MacLean, VA 22102

Dr. Keith T. Wescourt
FMC Corporation
Central Engineering Labs
1185 Coleman Ave., Box 580
Santa Clara, CA 95052

Dr. Douglas Wetzel
Code 12
Navy Personnel R&D Center
San Diego, CA 92152-6800

LCDR Cory deGroot Whitehead
Chief of Naval Operations
OP-112G1
Washington, DC 20370-2000

Dr. Heather Wild
Naval Air Development
Center
Code 6021
Warminster, PA 18974-5000

Dr. William Clancey
Stanford University
Knowledge Systems Laboratory
701 Welch Road, Bldg. C
Palo Alto, CA 94304

Dr. Michael Williams
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

A. E. Winterbauer
Research Associate
Electronics Division
Denver Research Institute
University Park
Denver, CO 80208-0454

Dr. Robert A. Wisher
U.S. Army Institute for the
Behavioral and Social
Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

Mr. John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Dan Wolz
AFHRL/MOE
Brooks AFB, TX 78235

Dr. George Wong
Biostatistics Laboratory
Memorial Sloan-Kettering
Cancer Center
1275 York Avenue
New York, NY 10021

Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152-6800

Dr. Joe Yasatuke
AFHRL/LRT
Lowry AFB, CO 80230

Mr. Carl York
System Development Foundation
181 Lytton Avenue
Suite 210
Palo Alto, CA 94301

Dr. Joseph L. Young
Memory & Cognitive
Processes
National Science Foundation
Washington, DC 20550

Dr. Steven Zornetzer
Office of Naval Research
Code 114
800 N. Quincy St.
Arlington, VA 22217-5000