CS - 76 - 1

PSMON: A Production System Monitor

by

Arthur Farley

TABLE OF CONTENTS

APPENDICES

ACKNOWLEDGEMENT

I gratefully acknowledge the assistance of Ms. Jana Barnett during the initial phase of PSMON implementation.

I. Introduction

Production systems are a means of simulating human cognitive processing. A production system (Newell and Simon, 1972; Newell, 1973) is an ordered list (ideally, a collection) of condition-action pairs (called rules) together with a set of available memories which hold symbolic contents. The condition part of each rule consists of an ordered list of conditions which are applicable to the contents of the available memories and which are satisfied or not satisfied by those contents. The action part of a rule is an ordered list of actions which are applicable to the contents of the active memories and which alter those contents.

The execution of a production system is cyclic in nature. During each cycle, the condition parts of the rules are applied, according to the specified list ordering, to the contents of the available memories until a condition part is found whose conditions are completely satisfied by the current contents. When a condition part is completely satisfied, its associated rule is said to 'fire', with its action half being executed. The cycle then starts again, until no condition part is satisfied or until the production system is directly deactivated by the action of a firing rule.

Available memories are distinguished by the classes of content they may hold and by their associated content management strategies. The production system being defined here (from now on called PSMON) has five different available memories currently implemented. Figure 1 shows three of the memories with arrows indicating allowable information flow. The first memory is the ENVIRONMENT. The ENVIRONMENT is a simplified, operational representation of external memory. It is represented in PSMON as an ordered list of elements. Once created, it remains constant during production system execution unless directly altered by rule action.

The second memory is STM (Short Term Memory). It is a simplified, operational representation of the limited active memory characteristic of the human cognitive processor. STM is an ordered list of a relatively small, constant number of informational units called chunks. Every chunk has two aspects, a mark and a content. The mark simplifies accessing; the content may be any symbolic structure which is considered to be a fundamental unit of representation

for a given application. When a new chunk is created, it is placed at the head of the STM list. When a chunk in STM is used during a rule firing, it is moved to the head of the list. Chunks are lost, or forgotten, by displacement from the end of the constant length list as new chunks are created. STM is the active memory. Whenever a condition of a rule which fires depends upon information in ENVIRONMENT or ITM, that information is automatically embodied in a new chunk in STM.

The third memory is ITM (Intermediate Term Memory). It serves the role of contextual memory for the system. Those chunks of STM which are of special interest or significance to a given production system (ie. a chosen move during problem solving) are transferred to ITM. This protects these chunks from complete loss when they are later displaced from STM. ITM is thus a list of chunks, transferred to ITM from STM by rule actions. A chunk may be recalled into STM by a rule condition during later processing.

One available memory which is not shown in Figure 1 is LTM (Long Term Memory). LTM contains the universe of symbols, symbol classes, interrelationships between symbols, and basic cognitive functions which are known to the system. The rules of the production system are written in terms of these known symbols, classes, relations, and functions. LTM underlies the whole system and thus it is not shown in Figure 1. LTM is defined anew for each given production system application. LTM is normally not affected by production system activity in PSMON. The other available memory not shown is that of rule, or immediate, memory. The conditions of a rule are satisfied by certain aspects of the available memories. Pointers are created to these aspects as conditions are satisfied. These pointers exist only during consideration of the rule's conditions, and, if it fires, during its action half. This allows actions to be naturally specified in terms of those aspects of available memory which led to their execution.

The following three sections will discuss three aspects of the PSMON system:

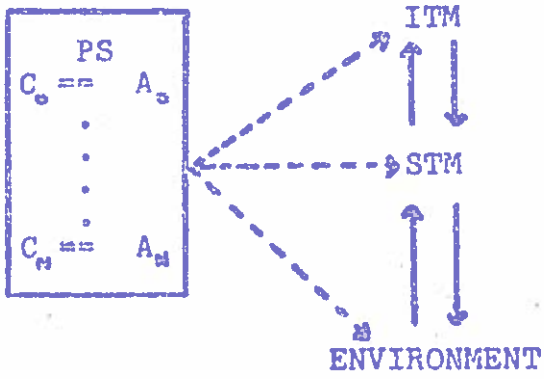
1) production system creation
2) available memory initialization
3) production system execution

Section V presents an annotated teletype protocol and output trace created by PSMON execution. The output generated by production system execution consists of a sequence of rule firings and available memory states. Section VI presents a brief description of a set of basic functions which are available to the user

as part of PSMON for the creation of production systems. Section VII presents a brief description of another set of functions, available to the user in an auxiliary file (LISTS.SNO), for dealing with list and attribute-value pair representations. Appendices present example production system and auxiliary LTM files.

PSMON is written in SNOBOL4 to run in the SITBOL system on the PDP-10. The user of PSMON is expected to have some familiarity with both SNOBOL4 and the SITBOL system. The user is expected to create SNOBOL4 program files defining needed condition and action functions for his given simulation application.

FIGURE 1 Production System and Available Memories



Solid arrows indicate normal information flow.

Broken arrows indicate conditions can apply to contents of all three memories.

II. Production System Creation

PSMON creates production systems by the processing of user-created production system files. In this section the syntax of production system files and the internal representation of production systems will be discussed.

Figure 2 shows a modified BNF definition of production system file syntax. Appendix A presents an example production system file.

FIGURE 2 Modified BNF specification of PS file syntax

```
‹ps-file› ::= ‹rule-defs› ‹ps-defs›
‹rule-defs› ::= ‹rule› | ‹rule› ‹rule-defs›
‹rule› ::= ‹name-line› ‹cond-lines› ‹arrow-line›
                ‹action-lines› ‹end-line›
‹name-line› ::= ‹rule-name› : ‹title›
‹rule-name› ::= R. ‹alphnums›
‹alphnums› ::= ‹letter› | ‹digit› | ‹alphnums› ‹letter› |
                ‹alphnums› ‹digit›
‹title› ::= any string
‹cond-lines› ::= ‹c-line› | ‹c-line› ‹cond-lines›
‹c-line› ::= ( ‹cond-name› : ‹specs-list› )
‹cond-name› ::= C ‹digit› | E ‹digit› | I ‹digit›
‹specs-list› ::= any condition function | any condition
                function .AND. ‹specs-list›
‹arrow-line› ::= ==›
‹action-lines› ::= ‹act› | ‹act› ‹action-lines›
‹act› ::= ( any action function )
‹end-line› ::= ENDRULE
‹ps-defs› ::= ‹ps-line› | ‹ps-line› ‹ps defs›
‹ps-line› ::= PS(' ‹ps-name› ',' ‹rule-list› ')
‹ps-name› ::= ‹letter› | ‹ps-name› ‹letter› | ‹ps-name› ‹digit›
‹rule-list› ::= ‹rule-name› | ‹rule-name› , ‹rule-list›
‹letter› ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
‹digit› ::= 1|2|3|4|5|6|7|8|9|0
```

A condition or action function consists of a call to a user, SNOBOL4, or PSMON defined function.

Some general characteristics of the syntax are worth noting. Blank lines are ignored by the system. Lines beginning with an asterisk are considered to be comments and are also ignored. A rule definition consists of several lines. Only one condition and one action can occur on a line. Condition, action, and arrow lines can have any number of initial tabs or blanks to allow formatting for readability, as illustrated in Appendix A.

The result of processing a rule definition is the creation of a rule data-structure, to which the rule name is then set. Figure 3 shows a graphic representation of the rule structure which would be generated for rule R.E 1 of Appendix A. The rule structure has four aspects: name, title, cond, and acts. The name and title are assigned on the basis of the first line of a rule definition. The title is an arbitrary string, used to describe briefly the semantics of the rule. The cond aspect is a list of conditions, one defined by each condition line. Each condition list element has three aspects: name, specs, and nextc. The name will be used during execution to determine to which available memory the condition is to be applied and will serve as a variable of immediate memory with its value being that aspect of the available memory satisfying the condition. The specs aspect is a list of condition functions associated with this condition. The nextc aspect points to the next condition element in the list for the given rule. The acts aspect of the role structure is a list of actions to be executed in list order when the rule is fired during production system execution.

After all rule definitions have been processed, the rule structures are next organized into production systems. This is accomplished by interpretation of the PS function specification(s), which occur at the end of the production system file. Note that more than one production system may be created from the set of rule structures. Each production system is named by the first argument of the function specification. Each production system will consist of an ordered list of rules, the order and rules being specified by the order of rule names occurring in the second argument of the function specification. Three production systems (named BPAM, LEARN, and ANSWER) are created by PSMON when processing the production system file of Appendix A.

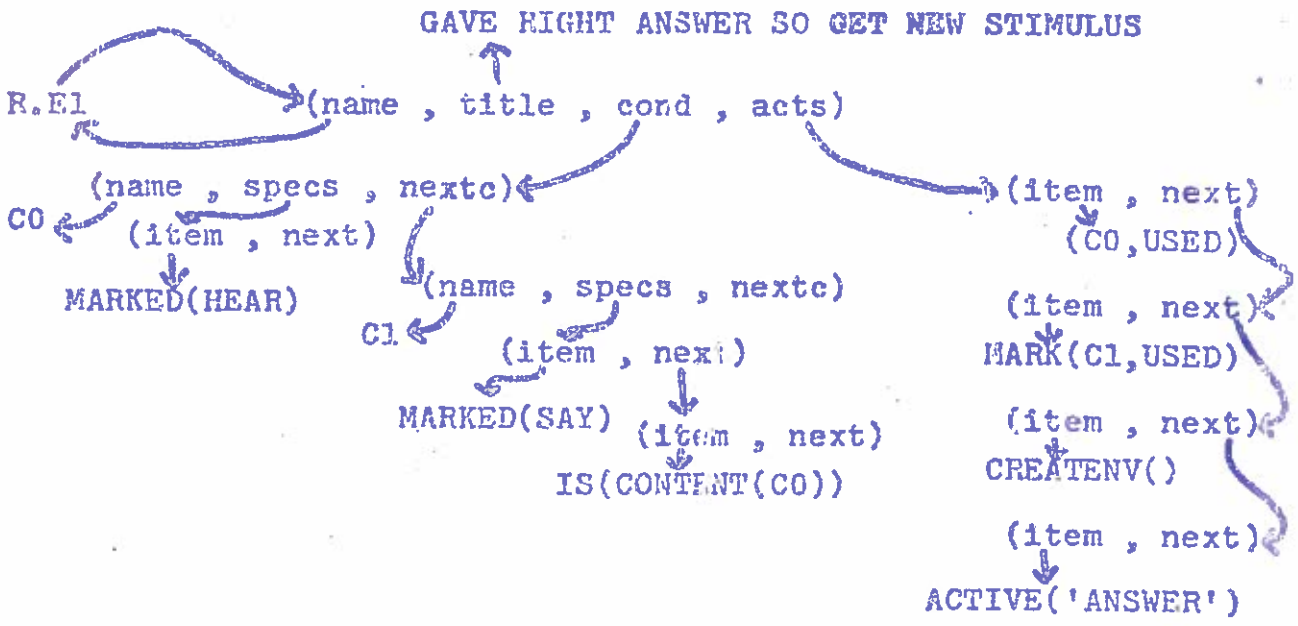
Production system creation is the first stage of processing performed by PSMON. PSMON asks the user at the teletype to provide names of production

system files which the user has created prior to running PSMON. See the annotated teletype protocol of Section V, which demonstrates this procedure.

FIGURE 3

```
!RULE  R.E1 #
R.E1:     GAVE RIGHT ANSWER SO GET NEW STIMULUS
          (C0:  MARKED(HEAR))
          (C1:  MARKED(SAY) .AND. IS(CONTENT(C0)))
                ==
                          (MARK(C0,USED))
                          (MARK(C1,USED))
                          (CREATENV())
                          (ACTIVE('ANSWER'))
ENDRULE
```

*RULE STRUCTURE FOR R.E1 *



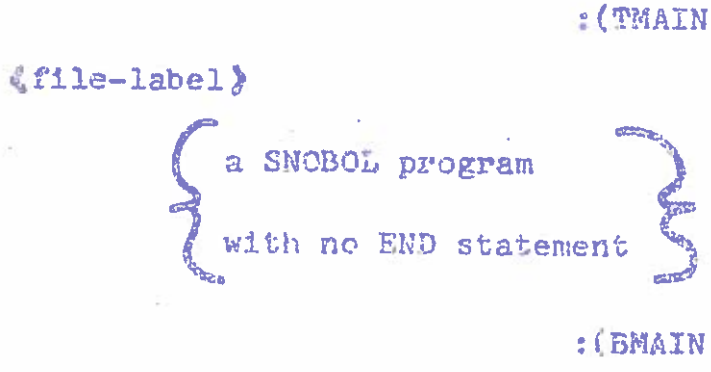
III. Available Memory Initialization

Before the production systems, created by the first phase of PSMON activity, can be executed, the available memories must be initialized and several output decisions must be made.

The user is asked whether he wants to produce teletype output in DEBUG mode or not. In DEBUG mode the current contents of STM are presented at the teletype after each system cycle. This mode is elected by responding with a 'T' and carriage return. If DEBUG mode is not elected, (by responding with a sole carriage return) only the name and title of the fired rule are presented at the teletype after each cycle.

LTM is the first available memory to be initialized. LTM is initialized by the execution of one or more auxiliary files. These much have been included before PSMON in the command string provided to the SITBOL system. (See beginning of the annotated teletype protocol of Section V.) Each auxiliary file must have the format specified by Figure 4. Appendix B presents an example LTM for a propositional reasoning application. The user is asked to input the name of the file label for each auxiliary file. Control is transferred there; the auxiliary file is executed, defining the symbols, classes, relations and functions specified by the SNOBOL program in the file. A symbol is a string which has itself as its value. A class is a string which has as its value a list of symbols as elements. Relations hold between symbols; for example, if OPPOSITE is a relation, UP could naturally be defined as the OPPOSITE of DOWN.

FIGURE 4 Auxiliary file format



Section VI lists several utility functions available to the user (as part of PSMON)for this purpose. User names must not conflict with names in PSMON. This can be guaranteed by starting all names with the letters U through Z. If this is not satisfactory, one should check the PSMON.SNO listing to avoid naming conflicts.

The user is then asked to input a file name to serve as output file for the trace produced by execution of the production system. A sole return at this point results in complete exit from the PSMON system.

The user is next asked to initialize STM. This must be done by providing a function call which is then evaluated by the system. Currently the only available utility function for this purpose is STM, which initializes STM to N null chunks when its second argument is missing, as illustrated in Section V. STM is considered to be an ordered list, but is represented in PSMON as a finite array (in case one wants to create new initialization functions). ITM is presently initialized to null automatically.

Finally the user is given the option of specifying an initial ENVIRONMENT. If the user elects to initialize the memory, he specifies a file name; otherwise the user responds with a sole return. TTY: indicates the information is to be input from the teletype. ENVIRONMENT is currently represented as a list of elements. Each line of input from the file or from the teletype is evaluated and becomes an element of the list.

IV. Production System Execution

After memory initializations and output decisions have been made, PSMON is prepared to execute a production system. The user is asked to provide the name of a production system to be activated. This is the name of one of the production systems created during processing of the production system files earlier. The general principles governing production system execution have been discussed in the introduction. Output from this execution consists of a trace of rule firings and resultant STM contents. Whenever ITM or the ENVIRONMENT is altered this change is also output. The output trace of Section V illustrates the form of the output trace. The mark aspect of an STM chunk is shown before the colons; the content aspect is presented after.

Several important aspects of production system execution by PSMON concern the way that conditions are satisfied by elements of the active memories. The initial letter of the name of each condition indicates which active memory is to serve as the source of satisfying contents. A condition named with a C indicates that a chunk of STM must satisfy the condition; an I indicates an element chunk of ITM; an E indicates an element of the ENVIRONMENT. When an element of the appropriate memory is found, the condition name is set to that element as an aspect of rule, or immediate, memory. An element of an available memory can satisfy only one condition of a rule.

Since the conditions of a rule are applied sequentially during production system executing the value of a previously established immediate memory element can be used in later conditions of the rule. This is illustrated by the conditions of rule R.E1 shown in Figure 3.

An important aspect of the condition satisfying process is that once a condition has been satisfied, the system will not back up and attempt to satisfy that condition with a different memory element when a later condition fails. This has several implications with regards to rule firing, one which can be illustrated with rule R.E1. Consider rule R.E1 applied to the following STM:

(HEAR :: CON) (SAY :: CON) (HEAR :: PAT)

The first chunk satisfies condition C0. The second chunk satisfies the second condition, so rule R.E1 fires. Consider now rule R.E1 applied to this STM:

(HEAR :: PAT) (HEAR :: CON) (SAY :: CON)

Again, the first chunk satisfies condition CØ. The second condition cannot be satisfied now, so rule R.E1 fails to fire. The system will not attempt to resatisfy condition CØ with the second chunk, which could cause rule R.E1 to fire. This convention simplifies the rule matching process, while arguments can be made for and against it being appropriate.

The last aspect of production system execution to be discussed is that of transfer of control between production systems. As can be seen in Appendix A, the action of a rule can activate another production system (by the ACTIVE function). PSMON maintains a production system name stack, the one named by the top of the stack being currently active. When the ACTIVE function is executed the name is pushed onto the stack. When the DEACT function is executed, or when no rule of the active production system is fired by current memory contents, the stack is popped. Note that more than one DEACT or ACTIVE function can occur in the action half of a single rule. Transfer of control becomes effective only at the beginning of a system cycle, the rule list of the newly activated production system being the one searched for condition satisfaction. If the production system name stack is empty at the beginning of any cycle, the end of production system execution is signalled. The output trace file is closed and control returns to the point where the user is asked to specify a trace file name. This allows the user to execute the production systems several times during one session without recreation of the systems. The teletype protocol of section V illustrates this option. If no file name is provided, PSMON execution ends and control is returned to SITBOL.

SECTION V

Example Protocol and Output Trace

This section presents teletype and file output created during a typical PSMON session. The example is a production system for propositional reasoning. Appendix B presents the auxiliary LTM file for this application. Appendix C presents the production system file.

ANNOTATED TELETYPE PROTOCOL

```
.RU TEN:SITBOL

*LOGIC,PSMON
```

To use PSMON the user must run the SITBOL system, which responds with an asterisk prompt for the command string. The user responds with a list of needed auxiliary LTM files, followed by PSMON. Note that no extension is necessary for .SNO files.

```
PRODUCTION SYSTEM FILE: LOGIC.RLS
PRODUCTION SYSTEM FILE:

DEBUG MODE :

LTM LABEL : TLOGIC
LTM LABEL :
```

The first phase of PSMON execution, creation of user production systems, requires the user to input the name(s) of existing, relevant production system file(s). All responses to questions from PSMON require a return. Here the user elects to input only one production system file name. Next, the user elects to not be in DEBUG mode by responding to the question with a sole return. The user is then asked to initialize LTM for this application. The response required is the program label at the top of the auxiliary file(s) , not the file name(s).

```
INPUT A FILE NAME FOR TRACE OF PS EXECUTION.
OR EXIT FROM SYSTEM WITH A SOLE RETURN.
TRACE FILE : LI.OUT

INITIALIZE STM WITH STM FUNCTION: STM(5)

INITIALIZE ENVIRONMENT : TTY:
INPUT NEW ENVIRONMENT; FINISH WITH SOLE *
NOT D
A
A OR D
IF A AND C THEN B
CONCLUDE B
*

INPUT THE NAME OF THE PRODUCTION SYSTEM
TO BE ACTIVATED FIRST: LOGIC
```

The user is then asked to perform a necessary sequence of initializing specifications prior to activating the first production system. The user must specify an output trace file name and initialize STM. The user is given the option of initializing ENVIRONMENT. A sole return would indicate the option is not taken; otherwise, a file name is provided. TTY:, as in this example, indicates that the ENVIRONMENT is to be initialized from the teletype. Finally, the production system is activated.

```
ACTIVATE LOGIC

R.100 FIRES        GET GOAL
R.3 FIRES          GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.3A1 FIRES        GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.3B1 FIRES        GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.2 FIRES          ELIMINATE IFTRUE, THUS PROVE GOAL
SAY: CONCLUSION IS PROVEN

DEACTIVATE LOGIC


INPUT A FILE NAME FOR TRACE OF PS EXECUTION.
OR EXIT FROM SYSTEM WITH A SOLE RETURN.
TRACE FILE : L2.OUT
```

Since DEBUG mode was not selected, a brief trace of production system activity is provided at the teletype, consisting only of the names and titles of the rules firing. After production system execution is completed, control returns to TSMON at the point where trace file specification must be made. Here the user elects to rerun the production system with a new ENVIRONMENT, as shown below.

```
INITIALIZE STM WITH STM FUNCTION: STM(3)

INITIALIZE ENVIRONMENT : TTY:
INPUT NEW ENVIRONMENT: FINISH WITH SOLE *
IF A OR B THEN C
NOT A
IF C THEN D
NOT B
CONCLUDE D
)*

INPUT THE NAME OF THE PRODUCTION SYSTEM
TO BE ACTIVATED FIRST: LOGIC
```

Note that STM is reinitialized, though it is again to contain null chunks, only these chunks are to be available, as opposed to five during the prior execution.

```
ACTIVATE LOGIC

R.100 FIRES       GET GOAL
R.3 FIRES         GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.3 FIRES         GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.3C1 FIRES       GENERATE NEW IFTRUE FROM CURRENT IFTRUE
R.1 FIRES         DETERMINE GOAL IS INCONSISTENT
SAY: CONCUSION IS INCONSISTENT

DEACTIVATE LOGIC


INPUT A FILE NAME FOR TRACE OF PS EXECUTION.
OR EXIT FROM SYSTEM WITH A SOLE RETURN.
TRACE FILE :
*^C
```

A brief trace is again presented at the teletype. The user is likewise again given the option of rerunning the production system with new initial specifications. The user elects to exit from PSMON by responding with a sole return. An asterisk appears, which is a prompt from the SITBOL system for a new command string. At this point the user may reenter PSMON for a new application with an appropriate command string. Here the user elects to return to the PDP-10 monitor with a control C. The session is completed by listing the trace files produced by production system execution. These file listings are presented on the following two pages.

```
.PRINT L1.OUT,L2.OUT
TOTAL OF 4 BLOCKS IN 2 FILES IN LPT REQUEST
```

FILE: L1.OUT

Output Trace Produced by
Logic Production
System

```
INITIAL STM
 STM:  () () () () ()

CURRENT ENVIRONMENT
NOT D
A
C OR D
IF A AND C THEN B
CONCLUDE B

ACTIVATE LOGIC

R,100 FIRES      GET GOAL

 STM:  (IFTRUE :: B) (NEW :: CONCLUDE B) () () ()

R,3 FIRES        GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: A AND C) (TRUE :: IF A AND C THEN B)
       (OLD :: B) (NEW :: CONCLUDE B) ()

R,3A1 FIRES      GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: C) (TRUE :: A) (OLD :: A AND C) (TRUE :: IF A AND C THEN B)
       (OLD :: B)

R,3B1 FIRES      GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: NOT D) (TRUE :: C OR D) (OLD :: C) (TRUE :: A)
       (OLD :: A AND C)

R,2 FIRES        ELIMINATE IFTRUE, THUS PROVE GOAL

 STM:  (SAY :: CONCLUSION IS PROVEN) (TRUE :: NOT D) (IFTRUE :: NOT D)
       (TRUE :: C OR D) (OLD :: C)

DEACTIVATE LOGIC
```

FILE: L2.OUT

Output Trace Produced by
Execution of Logic

```
INITIAL STM
 STM:  () () ()


CURRENT ENVIRONMENT
IF A OR B THEN C
NOT A
IF C THEN D
NOT B
CONCLUDE D


ACTIVATE LOGIC

R.100 FIRES      GET GOAL

 STM:  (IFTRUE :: D) (NEW :: CONCLUDE D) ()

R.3 FIRES        GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: C) (TRUE :: IF C THEN D) (OLD :: B)

R.3 FIRES        GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: A OR B) (TRUE :: IF A OR B THEN C) (OLD :: C)

R.3C1 FIRES      GENERATE NEW IFTRUE FROM CURRENT IFTRUE

 STM:  (IFTRUE :: B) (TRUE :: NOT A) (OLD :: A OR B)

R.1 FIRES        DETERMINE GOAL IS INCONSISTENT

 STM:  (SAY :: CONCLUSION IS INCONSISTENT) (TRUE :: NOT B)
         (IFTRUE :: B)


DEACTIVATE LOGIC
```

VI Basic Functions

In this section a brief description is given for each of the set of basic functions provided for the user as part of the PSMON system. Three classes of functions are provided: utility, condition, and action. The utility functions allow the user to initialize available memories. The condition and action functions are for use in the production systems written by the user.

One important aspect of function execution by SNOBOL must be discussed before the individual function descriptions are presented. This is that all arguments of a function are evaluated prior to function execution. This has important implications for the use of functions with string or name arguments. If the user wants to pass a literal string to function, that string must be quoted to prevent its evaluation. An example of this is seen in the PS functions of Appendix A. To pass an argument as a name, it must also be quoted. An example of this is also in Appendix A, in the arguments of the ACTIVE functions. To pass the name of the production system, and not the production system itself, one must quote the argument. An important note here, one must not make the name of a production system into a symbol of LTM or the production system is lost.

Utility Functions

STM(N,CNSTR) --creates an STM of N chunks and initializes them according to CNTSTR. CNTSTR is a string of the form: 'mark,content;.... ;mark,content'. Content is evaluated so a function can occur there. If CNTSTR is not specified, STM consists of N null chunks.
An example: STM (5,'HEAR,NO;SAY,YES') produces an STM, as follows:
(HEAR :: NO) (SAY :: YES) ( ) ( ) ( )

SYMBOLS(SYMSTR) --allows the user to define a set of symbols in LTM. SYMSTR is a string of the form: 'symbol, ...,symbol'. Each symbol is initialized with itself as its value, allowing each symbol to be used in an unquoted manner throughout a production system.

CLASSES(CLASS,SYMSTR) --allows the user to create symbol classes in LTM. CLASS is set to a list of symbols in SYMSTR. SYMSTR has the same form as in symbols. An example:
CLASSES ('BIRD', 'EAGLE, ROBIN, DODO')
where EAGLE, ROBIN and DODO are previously defined symbols.

RELATIONS(REL,PAIRSTR) --allows the user to create relations between symbols in STM. PAIRSTR has the form 'symbol, symbol;....;symbol,symbol'
An example:
RELATIONS ('OPPOSITE', 'UP,DOWN;DOWN,UP')
Relations are represented as tables in PSMON. Thus, later access is of the form OPPOSITE(UP), which would produce the value DOWN.

ACTION FUNCTIONS

MARK(CHUNK,MRK) --marks CHUNK with the mark MRK.

NEW(NAME,CONTENT) --creates a new chunk with the content of CONTENT evaluated, and sets NAME to the CHUNK, as a part of rule, or immediate, memory.
An example:
NEW ('C1', OPPOSITE (DOWN))

Note C1 is quoted as it is to be passed as a name. NAME should be of the form C<digit>

SAVE(CHUNK) --places CHUNK into ITM, as the last element in the ITM list.

DEACT( ) --deactivates the currently active production system, activating the next element on the production system name stack, if one exists.

ACTIVATE(PSNAME) --activates the named production system, pushing on the production system name stack.

LISTEN( ) --accepts input from the teletype and evaluates it, creating a new chunk at the head of STM with the resultant value as content and HEAR as mark. If the user wants to halt production system execution, he can type GOEND.

SAY(STRING) --types the string at the teletype. It also creates a new chunk at the head of STM with STRING as content and SAY as mark.

CREATENV(ENVFILE) --creates a new ENVIRONMENT. If ENVFILE is null it assumes TTY: and asks the user at the teletype to input a new one. If it is '*' it asks the user to specify a file name, which could be TTY: .

Condition Functions

IS(STR) --is satisfied if the chunk has as content the string STR.

MATCH(PATTERN) --is satisfied if the contents of the chunk is a string which is completely matched by the SNOBOL pattern PATTERN.

MATCH(PATTERN) --is satisfied if the contents of the chunk is a string whose initial part is matched by the SNOBOL pattern PATTERN.

MARKED(MARK) --is satisfied if the mark of the chunk is MARK.

NO('condition function') is satisfied if no chunk currently exists which satisfies the condition function in quotes.

Note that NO, MATCH, MATCH, and IS can be applied to elements of ITM and ENVIRONMENT also.

VII Auxiliary List Functions

In this section a brief description is given for each of a set of functions available to the user in an auxiliary LTM file LISTS.SNO. The functions provide utilities for an application requiring list or attribute-value pair lists. To use the functions discussed here the user would include LISTS before PSMON in the command string and type TLISTS when prompted for an LTM label.

Condition Functions

BEGINL(ARGL) --is satisfied if the current chunk has a list as content, and that list begins with the elements in ARGL. ARGL can be a list itself or a string representing a list of the form 'element, element,....,element'.

HASL(ARGL) --is satisfied if the current chunk has a list as content, and that list contains the elements in ARGL. ARGL can be a list or list string, as above.

SAMEL(L1,L2) --is satisfied if both L1 and L2 are lists and are the same lists.

MATAVP(AVLST) --is satisfied if the current chunk has an attribute-value pair list as content, and it has every attribute-value pair in AVLST. AVLST may be an a-v pair list or a string representing an a-v pair list of the form 'attribute,value; .....;attribute,value'.

Action Functions

NEWLST(ARGL) --creates a list from ARGL, a string representing a list, as above.

ATHEAD(LST,EL) --adds the element EL to the head of the list LST.

ATTAIL(LST,EL) --add the element EL to the tail of the list LST.

GETELN(LST,N) --return as value the Nth element of the list LST. Returns null if list shorter than N elements.

GETELA(LST,EL) --returns as value the element after EL from the list LST. Returns null if EL is not in the list or is the last element.

NEWAV(AVL) --creates a new a-v pair list from AVL, a string represent an a-v pair list, as above.

VIII Conclusion

The intent of this paper has been to introduce PSMON, a production system monitor written in SNOBOL4/SITBOL, both to potential users and to those merely interested in production systems as a theoretical model in psychology or as a computational system. Current research in production systems with PSMON centers about the creation of adaptive systems. An adaptive production system is one which inserts new rules or alters existent ones during its execution. Anyone interested in using the PSMON system should contact the author.

IX REFERENCES

[Newell, 1973] Newell, Allen, 'Production Systems: Models of Control Structures'; in Chase (ed.) Visual Information Processing; Academic: New York; 1973.

[Newell & Simon, 1972] Newell, A. and Simon, H., Human Problem Solving; Prentice Hall: Englewood Cliffs, N.J.; 1972.

```
3]  *    APPENDIX A.   EPAM.RLS
    *
    *       THREE PRODUCTION SYSTEMS, WHICH IN CONJUNCTION WITH ADAPT
    *       SIMULATE A VERSION OF EPAM.
    *
    *
    *.L1:   GET NEW FEATURE OF CURRENT STIMULUS
            (C3: MARKED(NEW))
            (E0: IS(CONTENT(C0)))
            (C1: MARKED(NEW))
            (E1: IS(CONTENT(C1)))
            (E2: MATCH(ASPECT))
            (C2: MARKED(HEAR))
                          ==>
                                  (MARK(E2,TRUE))
                                  (MARK(E1,TRUE))
                                  (MARK(E2,TRUE))
                                  (MARK(C2,USED))
                                  (MARK(C1,USED))
    ENDRULE
    *
    *
    *
    *.L1A:  GET NEW FEATURE OF CURRENT STIMULUS
            (C0: MARKED(NEW))
            (E0: IS(CONTENT(C0)))
            (E1: MATCH(ASPECT))
            (C1: MARKED(HEAR))
                          ==>
                                  (MARK(E2,TRUE))
                                  (MARK(E1,TRUE))
                                  (MARK(C2,USED))
    ENDRULE
    *
    *
    *
    *.L2:   CREATE CONDITION CHUNK
            (C0: MARKED(TRUE))
                          ==>
                                  (NEW('C1',ISCOND(CONTENT(C0),'E')))
                                  (MARK(C1,CND))
                                  (MARK(C2,USED))
    ENDRULE
    *
    *
    *
    *.L3:   CREATE ACTION CHUNKS AND ACTIVATE ADAPT
            (C2: MARKED(HEAR))
                          ==>
                                  (NEW('C1',DEACTACT()))
                                  (NEW('C2',SAYACT(CONTENT(C0))))
                                  (MARK(C2,USED))
                                  (MARK(C1,ACT))
                                  (MARK(C2,ACT))
                                  (DEACT())
                                  (ACTIVE('ANSWER'))
                                  (ACTIVE('ADAPT'))
    ENDRULE
    *
    *
```

```
*
*
*
R.E1:   SAVE RIGHT ANSWER SO GET NEW STIMULUS
        (C7: MARKED(HEAR));
        (C1: MARKED(SAY) .AND. (S(CONTEXT(C0)))
                        ==>
                                (MARK(C0.USED))
                                (MARK(C1.USED))
                                (CREATENV())
                                (ACTIVE('ANSWER'))
ENDRULE
*
*
*
R.E2:   SAVE WRONG ANSWER SO LEARN NEW RULE FOR ANSWER PS
        (C2: MARKED(HEAR))
        (C1: MARKED(SAY))
                        ==>
                                (MARK(C1.USED))
                                (ACTIVE('LEARN'))
ENDRULE
*
*
*
R.E3:   GET CORRECT ANSWER
        (C0: MARKED(SAY))
                        ==>
                                (LISTEN())
ENDRULE
*
*
*
R.E4:   GET NEW ENVIRONMENT
        (C0: SUCCEED)
                        ==>
                                (CREATENV())
                                (ACTIVE('ANSWER'))
ENDRULE
*
*
*
*
*
R.A100: DON'T KNOW ANSWER, SO SAY ?
        (E0: MATCH(ASPECT))
                        ==>
                                (SAY('?'))
                                (MARK(E0.TRUE))
                                (EXACT())
ENDRULE
*
*
*
*  -    DEFINE PRODUCTION SYSTEMS
*
*
PS('ANSWER','R.A100');
PS('REPAIR','R.E1,R.E2,R.E3,R.E4');
PS('LEARN','R.L1,R.L1A,R.L2,R.L3')
```

APPENDIX B. LOGIC.SNO

```
*
*       GO TO MAIN PSMONITOR
                                        :(TMAIN)
*
*
*
*
*       VARIABLES AND PROCESSES FOR LOGICAL REASONING
*
*
TLOGIC
*
*       DEFINE FUNCTIONS FOR LOGIC.RLS
*
        DEFINE('NOT(ST)')                       :(NLB)
NOT     NOT = ST
        NOT     POS(0) 'NOT '   =                       :S(RETURN)
        NOT = 'NOT ' NOT                                :(RETURN)


NLB
*
*       VALUES RELEVANT TO MODELLING OF LOGICAL
*       REASONING
*
*
        SYMBOLS('TRUE,IFTHLE,CLD')
        IF = 'IF '
        THEN = ' THEN '
        CONCLUDE = 'CONCLUDE '
        NOT = 'NOT '
        OR = ' OR '
        AND = ' AND '
*
*       DEFINE PATTERNS FOR LOGIC APPLICATION
*
        LET = SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        P = LET | LET   AND   LET | LET   OR   LET | NOT   LET
*
*
                                        :(BMAIN)
*
*    RETURN TO MAIN PSMONITOR
*
```

```
*
*       PRODUCTION SYSTEM FOR MODELLING LOGICAL REASONING
*
*
*
*
*,1:    DETERMINE GOAL IS INCONSISTENT
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('NOT(P1)'))
                        ==>
                                (MARK(E0,TRUE))
                                (SAY('CONCLUSION IS INCONSISTENT'))
                                (DEACT())
ENDRULE
*
*
*,2:    ELIMINATE IFTRUE, THUS PROVE GOAL
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('P1'))
                        ==>
                                (MARK(E0,TRUE))
                                (SAY('CONCLUSION IS PROVEN'))
                                (DEACT())
ENDRULE
*
*
*,2A1:  ELIMINATE IFTRUE, PROVE GOAL
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1 OR P . P2'))
        (E0: MATCH('P1'))
                        ==>
                                (MARK(E0,TRUE))
                                (SAY('CONCLUSION IS PROVEN'))
                                (DEACT())
ENDRULE
*
*
*,2A2:  ELIMINATE IFTRUE, PROVE GOAL
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1 OR P . P2'))
        (E0: MATCH('P2'))
                        ==>
                                (MARK(E0,TRUE))
                                (SAY('CONCLUSION IS PROVEN'))
                                (DEACT())
ENDRULE
*
*
*,2B1:  ELIMINATE IFTRUE, PROVE GOAL
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('P1 AND P'))
                        ==>
                                (MARK(E0,TRUE))
                                (SAY('CONCLUSION IS PROVEN'))
                                (DEACT())
ENDRULE
*
*
*,2B2:  ELIMINATE IFTRUE, PROVE GOAL
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('P AND P1'))
                        ==>
                                (MARK(E0,TRUE))
```

```
                                    (SAY('CONCLUSION IS PROVEN!'))
                                    (DEACT!))
ENDRULE
*
*
R.3:    GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('IF P . P2 THEN P1'))
                        ==>
                                    (MARK(E0,TRUE))
                                    (NEW('C1',P2))
                                    (MARK(C1,IFTRUE))
                                    (MARK(C0,OLD))
ENDRULE
*
*
R.3A1:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1  AND  P . P2'))
        (E0: MATCH('P1'))
                        ==>
                                    (MARK(E0,TRUE))
                                    (NEW('C1',P2))
                                    (MARK(C1,IFTRUE))
                                    (MARK(C0,OLD))
ENDRULE
*
*
R.3A2:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1 AND P . P2'))
        (E0: MATCH('P2'))
                        ==>
                                    (MARK(E0,TRUE))
                                    (NEW('C1',P1))
                                    (MARK(C1,IFTRUE))
                                    (MARK(C0,OLD))
ENDRULE
*
*
R.3B1:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('P1 OR P . P2'))
                        ==>
                                    (MARK(E0,TRUE))
                                    (NEW('C1',NOT(P2)))
                                    (MARK(C1,IFTRUE))
                                    (MARK(C0,OLD))
ENDRULE
*
*
R.3B2:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C0: MARKED(IFTRUE) .AND. MATCH('P . P1'))
        (E0: MATCH('P . P2 OR P1'))
                        ==>
                                    (MARK(E0,TRUE))
                                    (NEW('C1',NOT(P2)))
                                    (MARK(C1,IFTRUE))
                                    (MARK(C0,OLD))
ENDRULE
*
*
```

```
=,3C1:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C2: MARKED(IFTRUE) .AND. MATCH('P . P1 OR P . P2'))
        (E2: MATCH('NOT(P1)'))
                        ==>
                                (MARK(F2,TRUE))
                                (NEW('C1',P2))
                                (MARK(C1,IFTRUE))
                                (MARK(C2,OLD))
E:DRULE
*
*
=,3C2:  GENERATE NEW IFTRUE FROM CURRENT IFTRUE
        (C2: MARKED(IFTRUE) .AND. MATCH('P . P1 OR P . P2'))
        (E2: MATCH('NOT(P1)'))
                        ==>
                                (MARK(E2,TRUE))
                                (NEW('C1',P1))
                                (MARK(C1,IFTRUE))
                                (MARK(C2,OLD))
E:DRULE
*
*
=,5:    NO RULES APPLY FROM CURRENT MEMORY STATE. FAIL
        (C2: MARKED(IFTRUE))
                        ==>
                                (SAY('CAN NOT PROVE OR DISPROVE'))
                                (DEACT())
E:DRULE
*
*
=,1?3:  GET GOAL
        (E1: MATCH('CONCLUDE P . P1'))
                        ==>
                                (NEW('C0',P1))
                                (MARK(C0,IFTRUE))
E:DRULE
*
*
*
(**********************************************************************)
S('LOGIC',R,1,R,2,R,2A2,R,2B1,R,2B2,R,3,R,3A1,R,3A2,R,3B1,R,3B2,
```