A NEW DESIGN FOR SNIP
THE SNePS INFERENCE PACKAGE

Richard G. Hull

April 1986

SNeRG Technical Note No. 14

Department of Computer Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, New York 14260

A New Design for SNIP: the SNePS Inference Package

Richard G. Hull

Department of Computer Science
University at Buffalo
State University of New York
226 Bell Hall
Buffalo, New York 14260

1. Abstract

In this paper we describe a new design and partial implementation for SNIP, the inference package for the semantic network processing system SNePS [Shapiro 1979]. We begin with a brief description of the current version of SNIP, along with some of its shortcomings, followed by a description of the basic design elements of the new system. This is followed by a more detailed look at the algorithms used to implement the new design, and a look at the abstract data types used in this implementation. We also describe how this new design answers the above mentioned shortcomings of its predecessor. Finally, some suggestions are made for future enhancements to this latest implementation.

2. Introduction

The SNePS system has been in existence for several years, during which time it has been extensively modified, updated, extended, and simply corrected. It has gone through more than one translation to a new version of Lisp on a new machine. As the system became more complex, and modifications correspondingly harder, it was decided to

redesign the entire system from the bottom up, using modern techniques of data abstraction and software engineering.

The basic SNePS system consists of four major parts: (1) the "core" SNePS functions for building and finding information in the network, (2) a network pattern matcher, (3) an inference package, SNIP, which handles both forward and backward inference using rules which may be built into the network by the user, and (4) a simulated multi-processing system, MULTI, for controlling inference. At this writing, both the core functions and the pattern matcher have been re-written, along with the necessary abstract data types [Morgado 198x, Saks 1985]. In addition, a temporary interface has been provided between the new system and MULTI, which is a stand-alone system. This paper describes the new design and associated data types for SNIP. For purposes of distinction, we will refer to the current implementation of SNIP as SNIP79, and the new one as SNIP84.

3. A Brief Description of SNIP79

The nodes of SNePS represent unique intensional concepts, among which are included propositions. The rules for reasoning with these propositions are also intensional concepts, and consequently are also represented by network nodes. For a description of the types of rules which can be represented in SNePS, see [Shapiro & Rapaport, 1985]. When inference is initiated, whether it be forward inference or backward inference, MULTI processes are created to carry out the necessary passing and tallying of network information, as well as the calculations needed to determine when new knowledge can be inferred. Execution of these processes is controlled by MULTI. There are processes for collecting data, processes for initiating pattern matches, processes for

performing the actual rule calculations, and various processes for switching variable contexts, filtering out undesired information, and simply deciding which process to activate next (see [Martins, McKay & Shapiro, 1981]). The processes necessary to carry out a given inference make up an active connection graph which contains processes for each of the nodes, often more than one per node, and a few others not associated with any particular node. Modifications to the system have added various new processes, and the graph has gotten more and more complicated.

In addition, we may have several different instances of the same type of process working on various instances of the same node, and sharing of information between these processes is by no means guaranteed. So it was decided to simplify the design, in order to eliminate bugs that have arisen and to make future modifications easier.

4. The Basic Design of SNIP84

It is the intention of SNIP84 to treat inference as an activation of the network itself, rather than a compilation of the network into a distinct active connection graph of processes. To this end a much smaller set of processes is defined. SNIP84 processes are activations of nodes in the network -- there is one process attached to each proposition node. Thus, the types of processes required is limited to the types of nodes found in the network. These include general proposition nodes, rule nodes, and function nodes. In addition, there are processes to represent the system user -- processes which initiate the requested inference and collect the results. (For any particular inference there will be exactly one user process.) The node processes are attached directly to the network nodes, and communicate with each

other through "channels" which run parallel to rule arcs or between matched nodes. Following a somewhat object-oriented style, the processes send and receive various messages, and perform operations based solely on the information carried by incoming messages and the information stored at the node itself.

5. Message Types

There are four types of messages which will be sent between node-processes. Of these, just the first two have been implemented so far. They are:

REPORTS - messages containing substitutions which represent instances which have been determined to be true in the network. Reports also include whether a positive or negative instance is known, the name of the node sending the report, and the name of the actual node in the network which represents this instance.

Reports can be generated by forward inference, and as responses to requests initiated by backward inference.

REQUESTS - messages containing desired substitutions, and the necessary information to set up the channels through which reports of these instances can be sent.

STOP - a message telling a node that it should stop working on finding and sending out instances of itself, since they are no longer needed.

DONE - a message sent out by a node when it knows that it has already sent out all of the instances of itself that it possibly can.

6. Node Processes

In this section we describe the basic structure of each node process used by SNIP84. As mentioned previously, there are two major types of node processes: activations of (1) general propositions and (2) rules. Each process has a set of registers, which act as private memory and provide the mechanism for message passing. In addition, each process has associated with it a set of operations for dealing with the messages it receives.

6.1 Registers:

(1) General proposition nodes:

NAME: - the name of the process template used to create this process

NODE: - the node to which this process is attached

KNOWN-INSTANCES: - the collection of instances of this node (both positive and negative) which are known to be true

REPORTS: - reports received (used for implementation of message passing)

REQUESTS: - requests received (used for implementation of message passing)

INCOMING-CHANNELS: - the set of channels which will be feeding instance reports to this node

OUTGOING-CHANNELS: - the set of channels to which this node is to report instances that are discovered

PENDING-FORWARD-INFERENCES: - temporary storage for those reports which contained instances which had not been requested. These reports are only handled after all others have been taken care of.

(2) Rule nodes: in addition to all of the above registers, the processes attached to rule nodes also contain the following registers:

RULE-USE-CHANNELS: - the set of outgoing channels which travel parallel to rule arcs, and thus go to nodes which may be considered to be in consequent position with regard to the rule.

RULE-HANDLER: - the function used to determine when enough information is known to draw a new conclusion.

USABILITY-TEST: - a function to determine whether or not an instance of the rule can be used (based on the sign of the instance and

quantifier present).

(3) Every process also has two registers which are used solely to implement scheduling the process on the MULTI queue for the passing of the various kinds of messages. They are a (a) PRIORITY: register for scheduling the passing of reports ahead of requests, and (b) a QUEUES: register which keeps track of which parts of the queue (high-priority and/or low-priority) the process is in. The QUEUES: register is a temporary patch which should be eliminated when MULTI is rewritten to access its queue through abstract data type functions.

6.2 Operations:

The basic algorithm for all node processes is as follows:

```
if any reports have been received
  then handle them appropriately and terminate the process
else
  if any requests have been received
    then handle them appropriately
  if there any pending forward inferences
    then handle them appropriately
```

It should be noted that reports which are responses to deduction requests should be returned to the user as quickly as possible, while deduction requests themselves and forward inference reports should be propogated through the network in a parallel fashion. Thus we have a gradual spreading activation of the network interrupted only by prompt reports of specifically requested results. Toward this end, the MULTI process queue is divided into two parts: a "high priority" queue and a "low priority" queue. When a report is

sent to a process, that process is scheduled in the high-priority queue, whereas processes receiving requests are scheduled in the low-priority queue. Thus a node process which has received both requests and reports will be in both queues, and the above process termination after handling of reports does not mean that the requests will be missed. MULTI will take processes from the low-priority queue only when the high-priority queue is empty. Also, when reports are received which were not requested, and are thus actually forward inferences, they are placed in the PENDING-FORWARD-INFERENCES: register, and the node process is rescheduled in the low-priority queue. (See description below of report handling.)

Handling of reports, requests and forward inferences are as follows:

(1) General proposition nodes:

(a) Request handling:

Requests received by general proposition nodes come either from matched nodes or from rule nodes for which the proposition is in antecedent position. In both cases the requests are for valid instances of the proposition. The algorithm for handling requests is as follows:

```
install the requested outgoing-channel
if the node is asserted
  then send a report of this known instance
  else
    send any known instances through
    if a "wh-question" or no instances were sent
      then
```

if not already working on the desired instance
then
send requests to any dominating rules for which the node is in consequent position
if there are resources available and this request did not come through a "match channel"
then call match and send requests to all nodes found

Note: by match channel we mean a channel which connects two matched nodes (rather than nodes connected by rule arcs). Any request which arrives through such a channel must have already been sent to any other matching nodes, so we need not call match again in these cases.

Also: a "wh-question" is a request in which at least one of the variables in the request is not bound to a constant node. It corresponds to a question of "who?", "what?", "where?", etc.

(b) Report handling:

Reports received by general proposition nodes indicate substitutions for which the the proposition is known to be true. They are handled by the following algorithm:

go through the received reports one at a time,
eliminating those reports which contain instances which were already known
building new network nodes for previously unknown instances

broadcast each remaining report (representing a previously unknown instance) to all outgoing channels (placing those which pass none of the channel filters, and thus were not requested, into the PENDING-FORWARD-INFERENCES: register)
if the PENDING-FORWARD-INFERENCES: register is non-empty
then reschedule this node process on the low-priority queue

Note that we postpone the handling of forward inferences until after all responses to requests have been passed as far as possible. Thus, the node is rescheduled on the low-priority queue, where it will be activated only after all the processes on the high-priority queue have been handled.

(c) Forward inferences:

Forward inferences are simply reports for which there were no existing outgoing channels. They are handled as follows:

for each report in PENDING-FORWARD-INFERENCES:
send the report on to all rule nodes for which the proposition node is in antecedent position
if there are resources available and this report did not come through a "match channel"
then call match and send the report to all nodes found

(2) Rule nodes:

(a) Request handling:

Requests received by a rule node can be of two types. Since rules are propositions, they can receive requests for instances in the same way that any general proposition node might. These are handled exactly as described above.

The second type of request which may be received by a rule node is a request from a node in consequent position of the rule. Such a request would be for a substitution for which the consequent is known to be true. There are several situations which might occur.

(i) there are no free variables in the rule:

install the requested "rule use channel"
if the rule is asserted
then
if it is usable (based on quantifier)
then send requests to all nodes in antecedent
position which have not yet been requested
for this instance [1]

---

[1] Since a request will only come from a consequent if that consequent is not yet known, we can assume that not enough antecedents have sent reports to draw a conclusion. Thus rather than trying to apply the rule, we just make sure that all appropriate requests have gone out to the antecedents.

else try to establish the rule (as in the handling of requests to general proposition nodes)

(ii) there are free variables in the rule, but each is bound to a constant in the request:

```
install the requested rule use channel
if the desired rule instance is known
  then
    if it is usable (based on quantifier and sign)
      then pass the request back to antecedents
  else try to establish the requested rule instance
```

(iii) there are free variables in the rule, and at least one of them is not bound to a constant in the request

```
install the requested rule use channel
for each known rule instance with the correct substitution
  if it is usable (based on quantifier and sign)
    then pass requests back to the antecedents
try to establish additional rule instances
```

(b) Report handling:

Reports received by rule nodes can be of several types, and we have again chosen to describe each case separately. These report types are:

(i) reports of inferred instances of the rule

- requested as simple propositions
- requested for use of rule
- unrequested (forward inferences)

(ii) reports from antecedents

- requested
- unrequested (forward inferences)

In the following, references to "applying" the rule will be made. What is meant by this is that, for a given instance, we check the number of known antecedents, both positive and negative, and decide whether or not to draw conclusions. If enough antecedents are known, reports are sent to the consequents. If not, requests are sent to those antecedents which have not yet been sent requests for the instance under question.

(i) reports of inferred instances of the rule

for each such report,
send it out any outgoing channels whose filter it passes
if the reported rule instance is usable
then compare it to the filters of the rule use channels, and apply the RULE-HANDLER: for each filter passed
if the report passes no filters of either of the two above types of channels
then place it in PENDING-FORWARD-INFERENCES:S

(ii) reports from antecedents

for each such report,
if it is not the case that the rule is asserted but unusable
then
compare the report to the filters of the rule use channels
for each that it passes
if the node is asserted
then apply the RULE-HANDLER: to try to draw conclusions
else
if there exist applicable known-instances of the rule
then apply the RULE-HANDLER: for each known instance
if the report passes no rule use channel filters
then place it in PENDING-FORWARD-INFERENCES:

(c) Forward inferences:

Forward inferences which did not come from antecedents are handled similarly to forward inferences for general proposition nodes, with the addition that we also try to apply any newly inferred rule. That is, when a new rule instance is inferred, and thus built into the network, we set up rule use channels, with filters determined by the substitution for this new rule instance, to all nodes in consequent position, and send corresponding requests to all nodes in antecedent position.

Now we must also consider forward inferences

which consist of reports from nodes which are in antecedent position as far as the rule is concerned. For each of these we must first set up rule use channels to all nodes in consequent position. Then we proceed as if a request had just come in for the use of this instance of the rule. That is, we apply the algorithms described in 6.2-(2)-(a) above for request handling for rule nodes.

7. Data types of SNIP84

The primary concept behind the redesign of SNePS has been that of data abstraction. The design of SNIP84 has built upon the data types of this new implementation. For a complete description of the data types of SNePS, both the CoreSNePS and Match, see the forthcoming SNePS maintenance manual. In this section we will describe the additional data types defined for use by SNIP84. These data types fall into two categories: (1) those data types used in the implementation of the message passing facility of SNIP84, and (2) those data types used in constructing the messages themselves. This distinction is made less clear-cut by the fact that a request message consists solely of the channel being requested.

(a) Data types for constructing the message passing mechanism:

Two kinds of channels are used by SNIP84. They are:

(i) channels which connect matched nodes or run parallel to antecedent arcs (ant, &ant, arg)

(ii) "rule use" channels which run parallel to consequent arcs (cq, dcq, arg)

Of these, the first are the simplest, and will be discussed first. Such a channel is represented by the abstract data type called simply <channel>. A <channel> has the following five components:

1. a <filter> - for filtering out unwanted messages
2. a <switch> - for changing variable contexts between the source and destination nodes of the channel
3. an <mnoderep set> - containing information about unbuilt instances which may be part of the filter and/or switch
4. a <destination> - the place where the message sent through the channel will go
5. a <valve> - to turn the channel on and off (for future controlling of inference)

The following functions are defined for the data type <channel>:

RECOGNIZERS is.ch : <channel> --> <boolean>

CONSTRUCTORS make.ch : <filter> x <switch> x <mnoderep set> x <destination> x <valve> --> <channel>

SELECTORS
filter.ch : <channel> --> <filter>
switch.ch : <channel> --> <switch>
mnrs.ch : <channel> --> <mnoderep set>
destination.ch : <channel> --> <destination>
valve.ch : <channel> --> <valve>

```
TESTS          iscpen.ch       : <channel> --> <boolean>
               isclosed.ch     : <channel> --> <boolean>
               equivalent.ch   : <channel> x <channel>
                                      --> <boolean>
```

The following utility procedures are also defined:

```
open.ch         : <channel> -->
close.ch        : <channel> -->
```

which open or close a channel.

The structure which represents a <channel> is as follows:

A <channel> is simply a sequence of the above components, where the components are represented as follows:

```
<filter>        ::=  <substitution>

<switch>        ::=  <substitution>

<destination>   ::=  <node> | 'USER

<valve>         ::=  'OPEN | 'CLOSED
```

The data types <substitution> and <mnoderep set> are described in [Saks, 1985].

A rule use channel runs parallel to a consequent arc (cq,dcq,arg) from a rule node to a node in consequent position. Rule use channels are represented by the abstract data type <cq-channel>, which has the following components:

1. a <channel> - to actually carry the message
2. a <node set> - containing the set of nodes which are in antecedent position with respect to this consequent
3. a <rule-use-info-set> - containing information about known antecedents

The following functions are defined for the data type <cq-channel>:

CONSTRUCTORS make.cqch : <channel> x <node set> x <rule-use-info set> --> <cq-channel>

SELECTORS channel.cqch : <cq-channel> --> <channel>
ants.cqch : <cq-channel> --> <node set>
ruiset.cqch : <cq-channel> --> <rule-use-info set>

Again the underlying structure is that of a sequence of the above mentioned components. However, the <rule-use-info set> is a fairly complex structure itself, which we will now describe.

First:

<rule-use-info set> ::= { <rule-use-info> ... }

where sets are implemented using the data type <Set>. So the important part of the structure is the following definition:

<rule-use-info> ::= (<substitution>
<non-negative integer>
<non-negative integer>
<flagged-node set>)

where the substitution contains more complete bindings than the channel filter, such as bindings for the quantified variables, the first non-negative integer represents the number of antecedents known to be true, and the second represents the number of antecedents known to be false. The <flagged-node set> is a set of

<flagged-node> ::= (<node> . <truth-flag>)

where

<truth-flag> ::= 'TRUE | 'FALSE | 'UNKNOWN | 'REQUESTED

The nodes of the <flagged-node set> are the antecedents of the rule. Each <flagged-node> indicates whether that node is known to be true or false or unknown. For the unknown ones, a flag of 'REQUESTED indicates that a request has been sent to that antecedent already, but that no answering report has been received.

In addition to the outgoing channels, we also keep track of a set of incoming channels. These are represented by the data type <feeder>, which has the following components:

1. a <restriction> - the substitution (along with a required <mnoderep set>) which will be supplied by this channel

2. a <node> - the source node of this incoming channel
3. a <valve> - to turn the channel on/off

As mentioned above, a <restriction> is simply:

<restriction> ::= (<substitution> . <mnoderep set>)

(b) Data types for the messages themselves:

As mentioned previously, a <request> is simply defined as a <channel>. That is to say, a request message says to the receiving node: "Install the given channel and activate yourself to send out instances of yourself." On the other hand, a <report> must contain the following information:

1. a <substitution> - indicating the inferred instance
2. a <sign> - indicating whether the inferred instance is a positive or negative instance (i.e. known to be true or false)
3. a <signature> - the <node> sending the report
4. a <node> - the actual network node which represents the instance being reported

A report is again simply a sequence of these components. The data type <sign> is structured as follows:

<sign> ::= 'POS | 'NEG

The messages STOP and DONE have yet to be implemented.

8. Backward inference in SNIP84

Backward inference is initiated in SNePS by a call to the top level SNePSUL function deduce. This function takes as arguments an optional number field, indicating the number of desired results, followed by a "SNePSUL node description" (see the SNePS User Manual for details). The deduce function simply tbuilds a node satisfying the SNePSUL node description and passes this node (along with the number field if present) to deduce*. Deduce* creates a new "user" process to receive, tally and return the results of the deduction. This user process is described in more detail below. Deduce* then activates the temporary node sent to it by deduce, sends it a request (for a channel to the user process) and schedules the temporary node's process on the MULTI queue.

Note: since tbuild in the new implementation of SNePS tries first to find a node meeting the requirements, and only tbuilds one if no such node is found, the above references to a "temporary node" should actually be "the node returned by tbuild", which may be an already existing (permanent) node in the network.

Deduce* introduces three variables which are used non-locally by the system. They are:

DEDUCTION-RESULTS - which is side-effected by the USER process to contain all nodes which are reported to the USER process in response to the deduction request. It will not contain nodes built during intermediate stages of the deduction. When a deduction halts, the value of DEDUCTION-RESULTS: is

returned by deduce, via deduce*.

USER-PROCESS - which holds the process-id of the user process. This could easily be eliminated, since it is only accessed non-locally by one other function, but to make this change would require making the destination of a channel be the process rather than the node. I felt that for debugging purposes (during an ev-trace) it would be easier to trace the deduction with the node in the channel.

ADDED-NODES - which is side-effected during the entire inference process to contain all nodes built, even those at intermediate stages. This is currently not used by deduce, but is required for forward inference.

8.1 The user process

The user process is the one process which does not represent the activation of a node in the network, but rather the user who has asked for a deduction to be carried out. The user process contains the following registers:

REPORTS: - the reports received

DEDUCED-NODES: - the nodes deduced so far

TOTAL-DESIRED: - the total number of results desired

POS-DESIRED: - the number of positive results desired

NEG-DESIRED: - the number of negative results desired

POS-FOUND: - the number of positive results found so far

NEG-FOUND: - the number of negative results found so far

It also contains the utility registers PRIORITY: and QUEUES:, as described earlier.

The algorithm for the user process is:

```
go through the REPORTS: one at a time, and
    if the <node> of the report is not in DEDUCED-NODES:
      then
        update the appropriate XXX-FOUND: register
        put the node into DEDUCED-NODES:
DEDUCTION-RESULTS  <-- union of DEDUCTION-RESULTS and
                                        DEDUCED-NODES:
if the desired number of results has been found
  then suspend the inference
```

9. Forward inference in SNIP84

Forward inference is initiated by a call to the top level SNePSUL function add, which takes as its argument a SNePSUL node description. The node to be added is built into the network, and then passed to the function add* which checks to see if it is a constant molecular node. If so, an activation

process is created and attached, it is given a report of its own truth, and the process is placed on the MULTI queue. (Note that when this report is processed, the node will assert itself.)

Add* also introduces the variable ADDED-NODES, which was described previously. At the end of a forward inference, add* returns the value of ADDED-NODES to add, which then returns it at the top level.

10. Advantages of SNIP84

By attaching the inference processes to the actual nodes of the network, we avoid the confusion of a separate "active connection graph" of processes. Since we are using only one process per node, we will avoid unnecessary duplication of processes. Subsequent inferences use these same processes, and thus can make maximum use of previously derived information without searching far and wide for it. In SNIP79, a search of a global list of active "infer" processes was required.

The entire inference process in SNIP84 can be viewed as a spreading activation of the network. Information regarding known instances flows through the channels of the network to "neighboring" nodes as conclusions are drawn. This information flow is easy to trace, understand, and even control if necessary through the use of the valves on the channels. Forward inference becomes a natural part of the information flow.

One feature of SNIP (79 and 84) is that a deduction request will return both positive and negative answers. For example,

Given: All men are mortal.
All gods are immortal.
Socrates is a man.
Plato is a man.
Zeus is a god.

the response to the question "Who is mortal?" would be

Socrates is mortal.
Plato is mortal.
Zeus is not mortal.

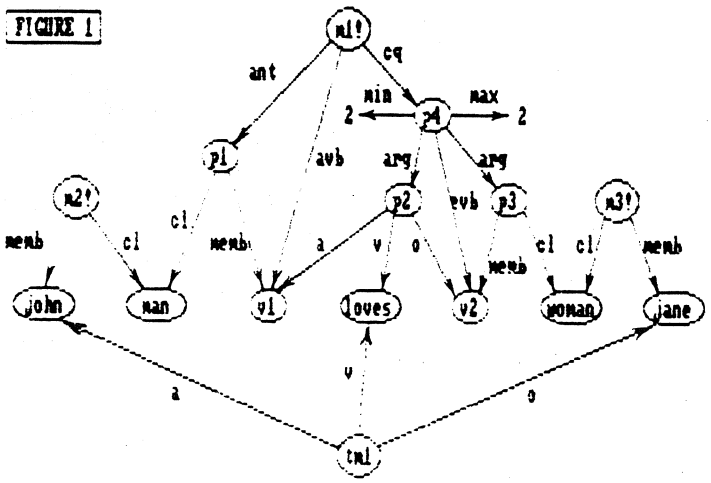
This is also important at intermediate stages of an inference when rules involving and-or, thresh, and numerical quantifiers are encountered. Both positive and negative instances of arguments can affect the conclusions drawn.

In SNIP84 reports contain a sign (POS or NEG) in addition to a substitution. This allows passing of both positive and negative instances in the same manner. The sign of each report can then be used by each rule node receiving the report, with no confusion even in the cases of positive and negative instances of negations.

The passing of signed reports also addresses one of the major bugs in SNIP79. In certain cases, existentially quantified rules were not used properly. Figure 1 shows the network representing the following propositions:

m1! : Every man loves some woman.
m2! : John is a man.
m3! : Jane is a woman.

FIGURE 1

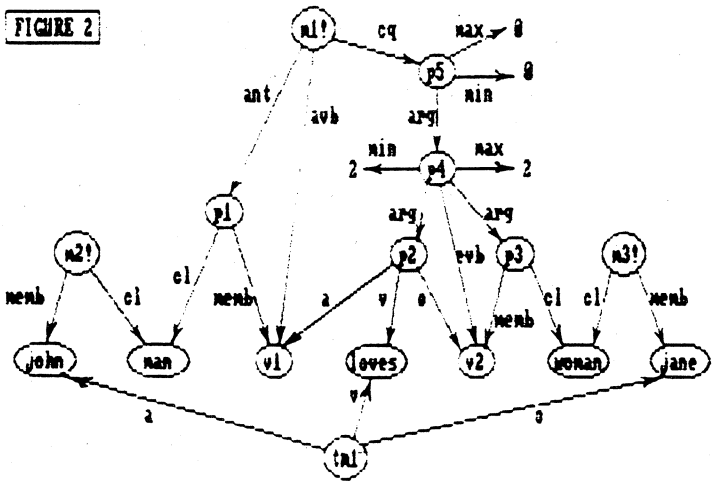


Clearly we cannot use rule m1! to answer the question "Does John love Jane?". In SNIP79, this inference was prevented by simply not allowing the match between the temporary node tm1 (built by the call to deduce) and the pattern node p2. The presence of the existential quantifier on the variable v2 blocked use of this rule, even though by themselves the nodes tm1 and p2 actually do match.

But now consider replacing rule m1! by the following (somewhat pessimistic) rule:

m1! : Every man loves **no** woman.

The resulting network is pictured in Figure 2.



This new rule can certainly be used to answer "Does John love Jane?" -- in the negative, of course. However, SNIP79 would reject the required match between tm1 and p2, as described above, and the rule would be ignored.

In SNIP84, the match would be allowed in both cases. Requests would be passed back to m1! and then to p1, and hence to m2!. The positive report sent back would generate a positive report from m1! to its consequent (p4 in Figure 1, p5 in Figure 2). In the first example, rule p4 would not be applied, since a positive instance of an existentially quantified rule cannot answer a specific question of the sort requested. This fact would be determined by applying p4's USABILITY-TEST:. However, in the second example the positive instance of p5 would generate a negative report to p4, which would be usable, and thus the proper conclusion would be drawn.

11. Remaining work

Several parts of SNIP84 are yet to be implemented. In particular, rule handler functions have been written for only or-entailment, and-entailment, and numerical entailment so far. Function nodes must also be addressed.

Other future work might include the following:

1. implement the STOP and DONE messages
2. implement resource limitation
3. make use of the valves on the current channels
4. incorporate belief revision

References

Martins, J., McKay, D.P., and Shapiro, S.C. [1981] "Bi-directional Inference", Technical Report No. 174, Department of Computer Science, SUNY at Buffalo.

McKay, D.P. and Shapiro, S.C. [1980] "MULTI - A LISP Based Multiprocessing System", Technical Report No. 164, Department of Computer Science, SUNY at Buffalo.

McKay, D.P. and Shapiro, S.C. [1981] "Using active connection graphs for reasoning with recursive rules", Proceedings of the Seventh International Joint Conference on Artificial Intelligence, William Kaufman, Los Altos, CA, pp. 368-374.

Saks, V.H. [1985] "A Matcher for Intensional Semantic Networks", unpublished technical report, Department of Computer Science, SUNY at Buffalo.

Shapiro, S.C. [1977a] "Representing and Locating Deduction Rules in a Semantic Network", Proc. Workshop on Pattern-Directed Inference Systems, SIGART Newsletter, 63, pp. 14-18.

Shapiro, S.C. [1977b] "Compiling Deduction Rules from a Semantic Network into a set of Processes", Abstracts of Workshop on Automatic Deduction, MIT, Cambridge, MA. (Abstract only)

Shapiro, S.C. [1979] "The SNePS semantic network processing system", in Associative Networks: The Representation and Use of Knowledge by Computers, N. V. Findler (ed.), Academic Press, New York, pp. 179-203.

Shapiro, S.C. and McKay, D.P. [1980] "Inference with Recursive Rules", Proceedings of the First Annual National Conference on Artificial Intelligence, William Kaufman, Los Altos, CA, pp. 151-153.

Shapiro, S.C., Martins, J., and McKay, D.P. [1982] "Bi-directional Inference", Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, MI, pp. 90-93.

Shapiro, S.C. [1986] "Symmetric Relations, Intensional Individuals, and Variable Binding", Technical Report No. 86-10, Department of Computer Science, SUNY at Buffalo.

Demonstration of backward inference, with chaining of rules:

```
* (define memb class obj prop)
(memb class obj prop)
CPU time : 0.83     GC time : 0.00

* (build avb $x
         ant (build memb *x class man)
         cq  (build memb *x class human))!
(m1)
CPU time : 3.52     GC time : 5.85

(m1!)
CPU time : 0.32     GC time : 0.00

* (build avb $y
         ant (build memb *y class human)
         cq  (build obj *y prop mortal))!
(m2)
CPU time : 3.15     GC time : 5.95

(m2!)
CPU time : 0.25     GC time : 0.00

* (build memb socrates class man)!
(m3)
CPU time : 0.93     GC time : 0.00

(m3!)
CPU time : 0.27     GC time : 0.00

* (build memb plato class human)!
(m4)
CPU time : 0.92     GC time : 0.00

(m4!)
CPU time : 0.27     GC time : 0.00

* (describe *nodes)

(m4! (memb plato) (class human))
(m3! (memb socrates) (class man))
(m2! (ant (p3 (memb (v2)) (class human)))
     (avb (v2))
     (cq (p4 (obj (v2)) (prop mortal))))
(m1! (ant (p1 (memb (v1)) (class man)))
     (avb (v1))
     (cq (p2 (memb (v1)) (class human))))

(m4! plato m3! socrates m2! p4 mortal p3 v2 m1! p2 human p1 man v1)
CPU time : 2.42     GC time : 0.00
```

```
* (deduce obj %who prop mortal)

I wonder if
((tp1 (prop (mortal)) (obj tv1)))

I wonder if
((p3 (class (human)) (memb v2)))

I know
((m4! (class (human)) (memb (plato))))

Since
((p3 (class (human)) (memb (v2 <-- plato))))

I infer
((p4 (prop (mortal)) (obj (v2 <-- plato))))

I wonder if
((p1 (class (man)) (memb v1)))

I know
((m3! (class (man)) (memb (socrates))))

Since
((p1 (class (man)) (memb (v1 <-- socrates))))

I infer
((p2 (class (human)) (memb (v1 <-- socrates))))

Since
((p3 (class (human)) (memb (v2 <-- socrates))))

I infer
((p4 (prop (mortal)) (obj (v2 <-- socrates))))

(m7! m5!)
CPU time : 69.62      GC time : 42.23

* (describe *nodes)

(m7! (prop mortal) (obj socrates))
(m6! (class human) (memb socrates))
(m5! (prop mortal) (obj plato))
(m4! (memb plato) (class human))
(m3! (memb socrates) (class man))
(m2! (ant (p3 (memb (v2)) (class human)))
     (avb (v2))
     (cq (p4 (obj (v2)) (prop mortal))))
(m1! (ant (p1 (memb (v1)) (class man)))
     (avb (v1))
     (cq (p2 (memb (v1)) (class human))))

(m7! m6! m5! m4! plato m3! socrates m2! p4 mortal p3 v2 m1! p2
 human p1 man v1)
CPU time : 3.10      GC time : 0.00
```

Nested rules -- demonstration 1:

```
* (describe *nodes)

(m4! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2 (member (v2)) (class woman)))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))
(m3! (member sue) (class woman))
(m2! (member jane) (class woman))
(m1! (member john) (class man))

(m4! p4 p3 loves p2 v2 p1 v1 m3! sue m2! woman jane m1! man john)
CPU time : 2.48     GC time : 0.00

* (deduce agent john verb loves object %whom)

I wonder if
((tp1 (object tv1) (agent (john)) (verb (loves))))

I wonder if
((p4 (cq (p3 (object v2) (agent (v1 <-- john)) (verb (loves))))
     (avb v2)
     (ant (p2 (class (woman)) (member v2)))))

I wonder if
((p1 (class (man)) (member (v1 <-- john))))

I know
((m1! (class (man)) (member (john))))

Since
((p1 (class (man)) (member (v1 <-- john))))

I infer
((p4 (cq (p3 (object v2) (agent (v1 <-- john)) (verb (loves))))
     (avb v2)
     (ant (p2 (class (woman)) (member v2)))))

I wonder if
((p2 (class (woman)) (member v2)))

I know
((m3! (class (woman)) (member (sue))))

Since
((p2 (class (woman)) (member (v2 <-- sue))))

I infer
((p3 (object (v2 <-- sue)) (agent (v1 <-- john)) (verb (loves))))
```

```
I know
((m2! (class (woman)) (member (jane))))

Since
((p2 (class (woman)) (member (v2 <-- jane))))

I infer
((p3 (object (v2 <-- jane)) (agent (v1 <-- john)) (verb (loves))))


(m7! m6!)
CPU time : 110.67      GC time : 67.08

* (describe *nodes)

(m7! (object jane) (agent john) (verb loves))
(m6! (object sue) (agent john) (verb loves))
(m5! (cq (p5 (object (v2)) (agent john) (verb loves)))
     (avb (v2))
     (ant (p2 (member (v2)) (class woman))))
(m4! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))
(m3! (member sue) (class woman))
(m2! (member jane) (class woman))
(m1! (member john) (class man))

(m7! m6! m5! p5 m4! p4 p3 loves p2 v2 p1 v1 m3! sue m2! woman
 jane m1! man john)
CPU time : 3.87      GC time : 0.00
```

Nested rules -- demonstration 2:

```
* (describe *nodes)

(m4! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2 (member (v2)) (class woman)))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))
(m3! (member sue) (class woman))
(m2! (member jane) (class woman))
(m1! (member john) (class man))

(m4! p4 p3 loves p2 v2 p1 v1 m3! sue m2! woman jane m1! man john)
CPU time : 2.62     GC time : 0.00

* (deduce agent %who verb loves object sue)

I wonder if
((tp1 (object (sue)) (agent tv1) (verb (loves))))

I wonder if
((p4 (cq (p3 (object v2) (agent v1) (verb (loves))))
     (avb v2)
     (ant (p2 (class (woman)) (member v2)))))

I wonder if
((p1 (class (man)) (member v1)))

I know
((m1! (class (man)) (member (john))))

Since
((p1 (class (man)) (member (v1 <-- john))))

I infer
((p4 (cq (p3 (object v2) (agent (v1 <-- john)) (verb (loves))))
     (avb v2)
     (ant (p2 (class (woman)) (member v2)))))

I wonder if
((p2 (class (woman)) (member (v2 <-- sue))))

I know
((m3! (class (woman)) (member (sue))))

Since
((p2 (class (woman)) (member (v2 <-- sue))))

I infer
((p3 (object (v2 <-- sue)) (agent (v1 <-- john)) (verb (loves))))

(m6!)
```

```
CPU time : 81.62     GC time : 54.92

* (describe *nodes)

(m6! (object sue) (agent john) (verb loves))
(m5! (cq (p5 (object (v2)) (agent john) (verb loves)))
     (avb (v2))
     (ant (p2 (member (v2)) (class woman))))
(m4! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))
(m3! (member sue) (class woman))
(m2! (member jane) (class woman))
(m1! (member john) (class man))

(m6! m5! p5 m4! p4 p3 loves p2 v2 p1 v1 m3! sue m2! woman
 jane m1! man john)
CPU time : 3.47     GC time : 0.00
```

Demonstration of use of recursive rules:

```
* (describe *nodes)

(m5! (parent c) (child d))
(m4! (parent b) (child c))
(m3! (parent a) (child b))
(m2! (&ant (p3 (parent (v3)) (child (v4)))
           (p4 (ancestor (v4)) (descendant (v5))))
     (avb (v3) (v4) (v5))
     (cq (p5 (ancestor (v3)) (descendant (v5)))))
(m1! (ant (p1 (parent (v1)) (child (v2))))
     (avb (v1) (v2))
     (cq (p2 (ancestor (v1)) (descendant (v2)))))

(m5! d m4! c m3! b a m2! p5 p4 p3 v5 v4 v3 m1! p2 p1 v2 v1)
CPU time : 3.47     GC time : 0.00

* (deduce ancestor a descendant d)

I wonder if
((tm1 (descendant (d)) (ancestor (a))))

I wonder if
((p3 (child v4) (parent (v3 <-- a))))

I wonder if
((p4 (descendant (v5 <-- d)) (ancestor v4)))

I wonder if
((p1 (child (v2 <-- d)) (parent (v1 <-- a))))

I know
((m3! (child (b)) (parent (a))))

I wonder if
((p3 (child v4) (parent v3)))

I wonder if
((p1 (child (v2 <-- d)) (parent v1)))

I know
((m5! (child (d)) (parent (c))))

Since
((p1 (child (v2 <-- d)) (parent (v1 <-- c))))

I infer
((p2 (descendant (v2 <-- d)) (ancestor (v1 <-- c))))

I know
((m4! (child (c)) (parent (b))))
```

```
Since
((p3 (child (v4 <-- c)) (parent (v3 <-- b))))
and
((p4 (descendant (v5 <-- d)) (ancestor (v4 <-- c))))

I infer
((p5 (descendant (v5 <-- d)) (ancestor (v3 <-- b))))

Since
((p3 (child (v4 <-- b)) (parent (v3 <-- a))))
and
((p4 (descendant (v5 <-- d)) (ancestor (v4 <-- b))))

I infer
((p5 (descendant (v5 <-- d)) (ancestor (v3 <-- a))))

I know
((m3! (child (b)) (parent (a))))


(m8!)
CPU time : 192.12      GC time : 112.88

* (describe *nodes)

(m8! (descendant d) (ancestor a))
(m7! (descendant d) (ancestor b))
(m6! (descendant d) (ancestor c))
(m5! (parent c) (child d))
(m4! (parent b) (child c))
(m3! (parent a) (child b))
(m2! (&ant (p3 (parent (v3)) (child (v4)))
           (p4 (ancestor (v4)) (descendant (v5))))
     (avb (v3) (v4) (v5))
     (cq (p5 (ancestor (v3)) (descendant (v5)))))
(m1! (ant (p1 (parent (v1)) (child (v2))))
     (avb (v1) (v2))
     (cq (p2 (ancestor (v1)) (descendant (v2)))))

(m8! m7! m6! m5! d m4! c m3! b a m2! p5 p4 p3 v5 v4 v3 m1! p2 p1 v2 v1)
CPU time : 4.20      GC time : 0.00
```

Demonstration of backward inference with numerical entailment:

```
* (describe *nodes)

(m6! (&ant (p1 (member (v1)) (class mathematician))
           (p2 (member (v1)) (class philosopher))
           (p3 (member (v1)) (class scientist)))
     (thresh 2)
     (avb (v1))
     (cq (p4 (member (v1)) (class multitalented))))
(m5! (member tom) (class scientist))
(m4! (member bill) (class scientist))
(m3! (member bill) (class philosopher))
(m2! (member fred) (class philosopher))
(m1! (member fred) (class mathematician))

(m6! p4 multitalented p3 p2 p1 v1-2 m5! tom m4! scientist m3! bill m2!
 philosopher m1! mathematician fred)
CPU time : 2.97      GC time : 0.00

* (deduce member %who class multitalented)

I wonder if
((tp1 (class (multitalented)) (member tv1)))

I wonder if
((p1 (class (mathematician)) (member v1)))

I wonder if
((p2 (class (philosopher)) (member v1)))

I wonder if
((p3 (class (scientist)) (member v1)))

I know
((m1! (class (mathematician)) (member (fred))))

I know
((m3! (class (philosopher)) (member (bill))))

I know
((m2! (class (philosopher)) (member (fred))))

Since
((p1 (class (mathematician)) (member (v1 <-- fred))))

and
((p2 (class (philosopher)) (member (v1 <-- fred))))

I infer
((p4 (class (multitalented)) (member (v1 <-- fred))))

I know
((m5! (class (scientist)) (member (tom))))
```

```
I know
((m4! (class (scientist)) (member (bill))))

Since
((p2 (class (philosopher)) (member (v1 <-- bill))))

and
((p3 (class (scientist)) (member (v1 <-- bill))))

I infer
((p4 (class (multitalented)) (member (v1 <-- bill))))


(m8! m7!)
CPU time : 135.63     GC time : 80.65

* (describe *nodes)

(m8! (class multitalented) (member bill))
(m7! (class multitalented) (member fred))
(m6! (&ant (p1 (member (v1)) (class mathematician))
           (p2 (member (v1)) (class philosopher))
           (p3 (member (v1)) (class scientist)))
     (thresh 2)
     (avb (v1))
     (cq (p4 (member (v1)) (class multitalented))))
(m5! (member tom) (class scientist))
(m4! (member bill) (class scientist))
(m3! (member bill) (class philosopher))
(m2! (member fred) (class philosopher))
(m1! (member fred) (class mathematician))

(m8! m7! m6! p4 multitalented p3 p2 p1 v1 2 m5! tom m4! scientist
 m3! bill m2! philosopher m1! mathematician fred)
CPU time : 3.43     GC time : 0.00
```

Demonstration of forward inference with nested rules:

```
* (describe *nodes)

(m1! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2 (member (v2)) (class woman)))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))

(m1! p4 p3 loves p2 woman v2 p1 man v1)
CPU time : 1.93     GC time : 5.92

* (add member john class man)

Since
((p1 (class (man)) (member (v1 <-- john))))

I infer
((p4 (cq (p3 (object v2) (agent (v1 <-- john)) (verb (loves))))
     (avb v2)
     (ant (p2 (class (woman)) (member v2)))))

I wonder if
((p2 (class (woman)) (member v2)))


(m3! m2!)
CPU time : 40.50     GC time : 30.35

* (describe *nodes)

(m3! (cq (p5 (object (v2)) (agent john) (verb loves)))
     (avb (v2))
     (ant (p2 (member (v2)) (class woman))))
(m2! (member john) (class man))
(m1! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))

(m3! p5 m2! john m1! p4 p3 loves p2 woman v2 p1 man v1)
CPU time : 2.78     GC time : 0.00

* (add member jane class woman)

Since
((p2 (class (woman)) (member (v2 <-- jane))))

I infer
((p3 (object (v2 <-- jane)) (agent (v1 <-- john)) (verb (loves))))
```

```
(m5! m4!)
CPU time : 34.98      GC time : 24.30

* (describe *nodes)

(m5! (object jane) (agent john) (verb loves))
(m4! (member jane) (class woman))
(m3! (cq (p5 (object (v2)) (agent john) (verb loves)))
     (avb (v2))
     (ant (p2 (member (v2)) (class woman))))
(m2! (member john) (class man))
(m1! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq
      (p4 (ant (p2))
          (avb (v2))
          (cq (p3 (verb loves) (agent (v1)) (object (v2)))))))

(m5! m4! jane m3! p5 m2! john m1! p4 p3 loves p2 woman v2 p1 man v1)
CPU time : 3.25      GC time : 0.00
```

Demonstration of simple backward inference with tracing showing the initiation of processes and contents of registers during execution:

```
* (describe *nodes)

(m2! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq (p2 (obj (v1)) (prop mortal))))
(m1! (member socrates) (class man))

(m2! p2 mortal p1 v1 m1! man socrates)
CPU time : 1.40     GC time : 0.00

* (deduce obj socrates prop mortal)

          ** new - user id - p00605 **

          ** new - non-rule id - p00606 **

           ***** entering processes:  p00606 *****
               NAME: non-rule
               NODE: tm1
               KNOWN-INSTANCES: nil
               REPORTS: nil
               REQUESTS: ((nil nil nil USER OPEN))
               INCOMING-CHANNELS: nil
               OUTGOING-CHANNELS: nil
               PENDING-FORWARD-INFERENCES: nil
               PRIORITY: LOW
               QUEUES: nil

I wonder if
((tm1 (prop (mortal)) (obj (socrates))))

          ** new - non-rule id - p00607 **

** initiate - non-rule id - p00607 initiated by - p00606 length of evnts - 1 **

           ***** leaving processes:  p00606 *****
               NAME: non-rule
               NODE: tm1
               KNOWN-INSTANCES: nil
               REPORTS: nil
               REQUESTS: nil
               INCOMING-CHANNELS: (((nil) p2 OPEN))
               OUTGOING-CHANNELS: ((nil nil nil USER OPEN))
               PENDING-FORWARD-INFERENCES: nil
               PRIORITY: LOW
               QUEUES: nil
```

```
          ***** entering processes:  p00607 *****
              NAME: non-rule
              NODE: p2
              KNOWN-INSTANCES: nil
              REPORTS: nil
              REQUESTS: ((((v1 . socrates)) nil nil tm1 OPEN))
              INCOMING-CHANNELS: nil
              OUTGOING-CHANNELS: nil
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: LOW
              QUEUES: nil

         ** new - rule id - p00608 **

** initiate - rule id - p00608 initiated by - p00607 length of evnts - 1 **

          ***** leaving processes:  p00607 *****
              NAME: non-rule
              NODE: p2
              KNOWN-INSTANCES: nil
              REPORTS: nil
              REQUESTS: nil
              INCOMING-CHANNELS: (((((v1 . socrates))) m2 OPEN))
              OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil tm1 OPEN))
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: LOW
              QUEUES: nil

          ***** entering processes:  p00608 *****
              NAME: rule
              TYPE: or-entailment
              NODE: m2
              KNOWN-INSTANCES: nil
              REPORTS: nil
              REQUESTS: ((((v1 . socrates)) nil nil p2 OPEN))
              INCOMING-CHANNELS: nil
              OUTGOING-CHANNELS: nil
              RULE-USE-CHANNELS: nil
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: LOW
              QUEUES: nil
              RULE-HANDLER: rule-handler.v-ent
              USABILITY-TEST: usability-test.v-ent

         ** new - non-rule id - p00609 **

** initiate - non-rule id - p00609 initiated by - p00608 length of evnts - 1 **

          ***** leaving processes:  p00608 *****
              NAME: rule
              TYPE: or-entailment
              NODE: m2
              KNOWN-INSTANCES: nil
              REPORTS: nil
              REQUESTS: nil
```

```
        INCOMING-CHANNELS: (((((v1 . socrates))) p1 OPEN))
        OUTGOING-CHANNELS: nil
        RULE-USE-CHANNELS: (((((v1 . socrates)) nil nil p2 OPEN)
                            (p1)
                            ((((v1 . socrates)) 0 0 ((p1 . REQUESTED))))))
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: LOW
        QUEUES: nil
        RULE-HANDLER: rule-handler.v-ent
        USABILITY-TEST: usability-test.v-ent

      ***** entering processes:  p00609 *****
        NAME: non-rule
        NODE: p1
        KNOWN-INSTANCES: nil
        REPORTS: nil
        REQUESTS: ((((v1 . socrates)) nil nil m2 OPEN))
        INCOMING-CHANNELS: nil
        OUTGOING-CHANNELS: nil
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: LOW
        QUEUES: nil

I wonder if
((p1 (class (man)) (member (v1 <— socrates))))

      ** new - non-rule id - p00610 **

** initiate - non-rule id - p00610 initiated by - p00609 length of evnts - 1 **

       ***** leaving processes:  p00609 *****
        NAME: non-rule
        NODE: p1
        KNOWN-INSTANCES: nil
        REPORTS: nil
        REQUESTS: nil
        INCOMING-CHANNELS: (((((v1 . socrates))) m1 OPEN))
        OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil m2 OPEN))
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: LOW
        QUEUES: nil

       ***** entering processes:  p00610 *****
        NAME: non-rule
        NODE: m1
        KNOWN-INSTANCES: nil
        REPORTS: nil
        REQUESTS: ((nil ((v1 . socrates)) nil p1 OPEN))
        INCOMING-CHANNELS: nil
        OUTGOING-CHANNELS: nil
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: LOW
        QUEUES: nil
```

```
I know
((m1! (class (man)) (member (socrates))))

** initiate - non-rule id - p00609 initiated by - p00610 length of evnts - 1 **

          ***** leaving processes:  p00610 *****
              NAME: non-rule
              NODE: m1
              KNOWN-INSTANCES: nil
              REPORTS: nil
              REQUESTS: nil
              INCOMING-CHANNELS: nil
              OUTGOING-CHANNELS: ((nil ((v1 . socrates)) nil p1 OPEN))
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: LOW
              QUEUES: nil

          ***** entering processes:  p00609 *****
              NAME: non-rule
              NODE: p1
              KNOWN-INSTANCES: nil
              REPORTS: ((((v1 . socrates)) POS m1 m1))
              REQUESTS: nil
              INCOMING-CHANNELS: (((((v1 . socrates))) m1 OPEN))
              OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil m2 OPEN))
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: HIGH
              QUEUES: nil

** initiate - rule id - p00608 initiated by - p00609 length of evnts - 1 **

          ***** leaving processes:  p00609 *****
              NAME: non-rule
              NODE: p1
              KNOWN-INSTANCES: ((((v1 . socrates)) . POS))
              REPORTS: nil
              REQUESTS: nil
              INCOMING-CHANNELS: (((((v1 . socrates))) m1 OPEN))
              OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil m2 OPEN))
              PENDING-FORWARD-INFERENCES: nil
              PRIORITY: HIGH
              QUEUES: nil

          ***** entering processes:  p00608 *****
              NAME: rule
              TYPE: or-entailment
              NODE: m2
              KNOWN-INSTANCES: nil
              REPORTS: ((((v1 . socrates)) POS p1 m1))
              REQUESTS: nil
              INCOMING-CHANNELS: (((((v1 . socrates))) p1 OPEN))
              OUTGOING-CHANNELS: nil
              RULE-USE-CHANNELS: (((((v1 . socrates)) nil nil p2 OPEN)
                                  (p1)
                                  ((((v1 . socrates)) 0 0 ((p1 . REQUESTED))))))
```

```
            PENDING-FORWARD-INFERENCES: nil
            PRIORITY: HIGH
            QUEUES: nil
            RULE-HANDLER: rule-handler.v-ent
            USABILITY-TEST: usability-test.v-ent
```

Since
((p1 (class (man)) (member (v1 <— socrates))))

I infer
((p2 (prop (mortal)) (obj (v1 <— socrates))))

** initiate - non-rule id - p00607 initiated by - p00608 length of evnts - 1 **

```
        ***** leaving processes:  p00608 *****
            NAME: rule
            TYPE: or-entailment
            NODE: m2
            KNOWN-INSTANCES: nil
            REPORTS: nil
            REQUESTS: nil
            INCOMING-CHANNELS: (((((v1 . socrates))) p1 OPEN))
            OUTGOING-CHANNELS: nil
            RULE-USE-CHANNELS: (((((v1 . socrates)) nil nil p2 OPEN)
                                (p1)
                                ((((v1 . socrates)) 1 0 ((p1 . TRUE))))))
            PENDING-FORWARD-INFERENCES: nil
            PRIORITY: HIGH
            QUEUES: nil
            RULE-HANDLER: rule-handler.v-ent
            USABILITY-TEST: usability-test.v-ent

        ***** entering processes:  p00607 *****
            NAME: non-rule
            NODE: p2
            KNOWN-INSTANCES: nil
            REPORTS: ((((v1 . socrates)) POS m2 nil))
            REQUESTS: nil
            INCOMING-CHANNELS: (((((v1 . socrates))) m2 OPEN))
            OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil tm1 OPEN))
            PENDING-FORWARD-INFERENCES: nil
            PRIORITY: HIGH
            QUEUES: nil
```

** initiate - non-rule id - p00606 initiated by - p00607 length of evnts - 1 **

```
        ***** leaving processes:  p00607 *****
            NAME: non-rule
            NODE: p2
            KNOWN-INSTANCES: ((((v1 . socrates)) . POS))
            REPORTS: nil
            REQUESTS: nil
            INCOMING-CHANNELS: (((((v1 . socrates))) m2 OPEN))
            OUTGOING-CHANNELS: ((((v1 . socrates)) nil nil tm1 OPEN))
            PENDING-FORWARD-INFERENCES: nil
```

```
        PRIORITY: HIGH
        QUEUES: nil

    ***** entering processes:  p00606 *****
        NAME: non-rule
        NODE: tm1
        KNOWN-INSTANCES: nil
        REPORTS: ((((v1 . socrates)) POS p2 m3))
        REQUESTS: nil
        INCOMING-CHANNELS: (((nil) p2 OPEN))
        OUTGOING-CHANNELS: ((nil nil nil USER OPEN))
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: HIGH
        QUEUES: nil

** initiate - user id - p00605 initiated by - p00606 length of evnts - 1 **

    ***** leaving processes:  p00606 *****
        NAME: non-rule
        NODE: tm1
        KNOWN-INSTANCES: ((((v1 . socrates)) . POS))
        REPORTS: nil
        REQUESTS: nil
        INCOMING-CHANNELS: (((nil) p2 OPEN))
        OUTGOING-CHANNELS: ((nil nil nil USER OPEN))
        PENDING-FORWARD-INFERENCES: nil
        PRIORITY: HIGH
        QUEUES: nil

    ***** entering processes:  p00605 *****
        NAME: user
        REPORTS: ((((v1 . socrates)) POS tm1 m3))
        DEDUCED-NODES: nil
        TOTAL-DESIRED: nil
        POS-DESIRED: nil
        NEG-DESIRED: nil
        POS-FOUND: 0
        NEG-FOUND: 0
        PRIORITY: HIGH
        QUEUES: nil

    ***** leaving processes:  p00605 *****
        NAME: user
        REPORTS: nil
        DEDUCED-NODES: (m3)
        TOTAL-DESIRED: nil
        POS-DESIRED: nil
        NEG-DESIRED: nil
        POS-FOUND: 1
        NEG-FOUND: 0
        PRIORITY: HIGH
        QUEUES: nil

(m3!)
CPU time : 37.63     GC time : 18.33
```

```
* (describe *nodes)

(m3! (prop mortal) (obj socrates))
(m2! (ant (p1 (member (v1)) (class man)))
     (avb (v1))
     (cq (p2 (obj (v1)) (prop mortal))))
(m1! (member socrates) (class man))

(m3! m2! p2 mortal p1 v1 m1! man socrates)
CPU time : 1.65     GC time : 6.10
```