CMU-CS-79-130

A SEMANTIC NETWORK OF PRODUCTION RULES IN A SYSTEM FOR DESCRIBING COMPUTER STRUCTURES

Michael D. Rychener

28 June 1979

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, PA 15213

This paper is an expanded version of one presented at the Sixth International Joint Conference on Artificial Intelligence, Tokyo, August, 1979.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

A Semantic Network of Production Rules in a System for Describing Computer Structures

Abstract. A novel implementation of the basic mechanisms of a semantic network is presented. This constitutes a merging, in terms of the underlying language architecture, of a powerful problem-solving mechanism, production-rule systems, with a proven representation formalism. Details are presented on the most basic aspects of the network, namely on representing nodes and on mechanisms for their access. Commands for definition, modification, and search-based displays of network information are discussed. The relations of the network are divided into six groups: taxonomic, structural, functional, descriptive, means-ends, and physical. The "further specification" relation is put forward as an improvement over concepts such as "ISA", superset, instantiation, individuation, and the type-token distinction. The importance of uniformly representing methods and network as rules, and the importance of distinguishing temporary from permanent states are discussed. Since the system is rule-based, it includes a simple but powerful augmentation capability, embodied in a language for expressing methods. Though evidence is not provided here for advanced semantic net capabilities, there is sketched a production system position on a number of relevant issues for those capabilities. The domain of application is the symbolic description and manipulation of computer structures at the PMS (processor-memory-switch) level. The system will ultimately be used for computer-aided design activities.

Keywords: production systems, semantic networks, representation of knowledge, knowledge acquisition, production rules, rule-based systems, symbolic description, understanding systems, computer-aided design.

1. Symbolic Description and Manipulation as a task for AI

1.1. Motivation and research context

One aspect of computer-aided design is the manipulation of symbolic descriptions of physical systems. Problems in this area have been discussed by Eastman [5] and by Sussman [23, 24, 25]. Other artificial intelligence (AI) research has discussed mechanisms that may be applicable to design systems while maintaining a general viewpoint and vocabulary, eg, Rieger's Commonsense Algorithms [17] and Moore's MERLIN [15, 16]. The present research aims to deal with the following problem areas:

1. Describing objects that are the basic components in a system.

2. Organizing those components into structures on the basis of their internal attributes and external relationships.

3. Establishing hierarchies of components and structures ranging from abstract, general ones to various versions of concrete realizations (instantiations) of the abstract components.

4. Comparing structures and putting structures into correspondence with each other (mapping).

5. Analyzing structures and determining effects of changes within them.

6. Synthesizing structures from elementary components, with the aim of fulfilling functional specifications.

7. Coordinating multiple viewpoints or descriptive hierarchies for the same system.

8. Searching in a design space of related systems.

9. Simulating system behavior symbolically, to ascertain dynamic properties lacking exact analytic formulation.

The system presented here addresses the first three topics, with a clear intention to proceed soon to others.

An approach to these problems would be applicable to a wide range of systems: buildings, software systems, chemical processes, cities, etc. The techniques are not restricted to the analysis and synthesis problems of the design area: they could provide a central mechanism for diagnostic, tutorial, explanatory, and theory explication systems. Such systems have been called "understanding systems" by Moore and Newell [16]. This is a virtually unexplored application area for AI research, and it appears to be rich in significant problems. Design is a ubiquitous phenomenon in science and engineering (see, for instance, Simon [22]). AI researchers need to gain more exposure to the basic concepts, and should try harder to develop a working familiarity and vocabulary with it.

Computer structures can be explored at many levels: logic circuitry, register-transfer level, instruction-set processor level, and processor-memory-switch (PMS) level. The PMS level is the subject domain of the IPMSL (Instructable PMS Language) system. The basis is that of Bell and Newell [3]. In particular, the following abstract components are considered:

C Computer	L Link
P Processor	D Data-operation
M Memory	N Network
S Switch	X External (non-digital)
K Controller	T Transducer
H Port	

Knudsen's [12] work in the PMS area is a direct predecessor to the present effort, with strong influences from Barbacci and Siewiorek [2].

1.2. A potential specific problem domain

A particularly important application of IPMSL is the hardware configuration problem, which focuses on evaluating specific configurations of peripheral and other devices for some computer.[1] Such a problem is posed by listing a set of components that a user or customer requires for his facility. IPMSL must then relate the customer's specification to known workable configurations and determine: which components are missing from it, if any (eg, due to hardware or software prerequisites); which component combinations give rise to the need for further hardware (as a result of component interactions); which component combinations give rise to the possibility of failure or low reliability; and whether the concrete configuration is a suitable match with the user's computing requirements. This list of capabilities is not at all exhaustive.

Consideration of this problem arises from the knowledge that this problem is a serious one for most mini-computer manufacturers. Such computers have a great deal of configurational flexibility, and the demands placed on such computers by customers tend to exploit that flexibility. This leads to problems at several locations in the computer delivery bureaucracy: at the sales level, where costs of final systems have to be quoted to a customer (the manufacturer usually absorbs the cost of mistakes here, which can be sizeable); at an engineering and assembly level, where configurations are built and tested out initially; and at the installation level, where there can be problems of supplying in timely fashion all the various parts and devices.

Though alternative formulations are possible, the representation of knowledge as a semantic network, along with its accompanying "technology", is the basis for IPMSL. A recent survey of this area of AI is given by Brachman [4]. The following sections will discuss: the basic design of the network, including its novel use of the production system architecture; the approach taken to building the system; peculiar features of semantic networks when implemented as production rules; the position of the present system with respect to a number of traditional semantic network issues; and finally an evaluation of its significance.

[1]Much of the specific knowledge needed for this domain is being developed by John McDermott.

2. The IPMSL system: basic semantic net representations

IPMSL (Instructable PMS Language) is a set of production rules for building semantic net structures in response to user commands.[1] IPMSL starts out as a set of "procedural" productions that perform the basic net-building operations. The net itself is composed of "declarative" productions whose contents are the net's facts. There are a few "procedural" productions devoted to constructing the "declarative" ones, on command from the user and also as a result of other processing. Both kinds of productions are interpreted according to the same basic recognize-act cycle. The user sees the information he enters being structured into a semantic network, while underneath, that information is stored as, and manipulated by, rules. The uniformity of representation appears to have some advantages, which will be discussed further below. In order to present the details of how a net looks as production rules, we first review the concept of production system architecture, discuss how it is used in IPMSL, and mention the underlying language being used. Then we proceed to define the network and the operations that can be performed at the user level of IPMSL.

2.1. Underlying problem-solving architecture

A production system architecture (PSA) [7, 19, 27] has four components: Working Memory (WM), Production Memory (PM), the Recognize-Act Cycle (RAC), and the Conflict Resolution procedure (CR). WM contains the dynamic state of the system, and is the "blackboard" where production rules make tests of patterns and make changes representing additions or modifications to the knowledge state. WM elements are transitory, so WM cannot be used for long-term storage of facts. PM contains the set of production rules, and is a permanent memory. RAC consists of an infinite repetition of three steps: testing the patterns (left-hand sides) of all rules in PM, to see which ones are true of the current WM (recognition step); deciding among the true rules which one(s) are to be executed on this cycle, which is done by CR; and executing the actions of the chosen rule(s) (action step). The result of an action step is a new WM state, and control returns to the recognition step, where a new set of rules can now become true, and so on.

The production system architecture of IPMSL is OPS2 [7]. OPS2 is a LISP-based system whose WM is a set of list structures (S-expressions). Each WM element has an associated "time tag", and OPS2 periodically deletes from WM the oldest elements, a strategy which amounts to placing an upper bound on the size of WM. IPMSL currently has a "retirement"

[1]This paper is primarily concerned with a basic version of IPMSL. In the process of revising and submitting it, IPMSL has almost doubled in size, as described more fully at the end of the paper. In several places below, specific improvements made to the basic IPMSL will be pointed out.

age of 1000 WM transactions. PM in OPS2 is an unstructured set of productions. That is, the rules are not organized into subroutines. Rather, all rules take part equally in each recognition. CR in OPS2 is done by considering the following criteria in order: (1) Refraction: repeated execution of the same rule using the same true pattern of WM elements is suppressed; (2) WM Recency: rules are preferred that use more recent WM elements (sorting is lexicographic by "time tag"); (3) Special Case: rules that use more WM elements, or that test such elements more precisely, are preferred; (4) PM Recency: the more recently created rule is preferred; (5) Arbitrary: if there is still a conflict, a rule instantiation is selected arbitrarily. This particular set of CR principles is quite instrumental in providing control adequate for the IPMSL task.

The implementation language for IPMSL uses the OPS2 architecture but extends its usefulness for this task by changing its external appearance. The extension is named OPS3RX. The primary capability afforded by OPS3RX is the ability to represent data elements as sets of attribute-value pairs. This allows patterns to select more flexibly various subparts of elements, and enhances the expressiveness and readability of the production rules. OPS3RX makes things look much more like "schemas" or "frames", and it is expected that further development of the language will continue in this direction. In fact, production rules seem to be an ideal way of expressing procedures in a frame-based system. Given the retirement limit of 1000 mentioned above for WM elements, there are usually about 100 attribute-value sets in WM, each containing about five attribute-value pairs.

The architecture as it is used in IPMSL has a rather novel appearance, at least among semantic network implementations. Rules are used to implement both the interpreter of the network and the network itself. WM serves both to hold onto various processing goals and to accumulate temporary network structures. A network rule contains the facts for only one node in the network, including pointers to other nodes. When some goal[1] requires the expansion of the net in WM along some direction (eg, in order to search for some piece of information), a subgoal appropriate to going in that direction is formed, resulting in a rule's execution. This causes appropriate structures to be hooked into existing WM structures, both by creating new WM nodes and by using pattern matching to detect appropriate linking

[1]A goal is a WM object consisting of attribute-value pairs that describe what it aims at and how it is progressing.

locations. How this works in detail will be explained shortly.[2]

2.2. Basic network design

IPMSL divides its knowledge about computer structures, and about itself, into six subnets, each containing a particular sort of knowledge:

- Structural. Parts, Partof, and Coparts relations. A component of a computer can be considered as a black box, or as an assemblage of known parts. Components are related to each other at the same structural- hierarchy level with the Coparts relation. In a computer, such a connection is used as a data or control path between components. Parts and Partof are converses of each other, and both are stored explicitly.

- Taxonomic. FSs and FSof relations. "FS" stands for "further specification", as used in the Merlin system [15, 16]. An object is an FS of another if it can be viewed as the other object with some additional characteristics. More precisely, the set of attributes in the description of the FS is a superset of those that describe the parent, and the values of those attributes may be modifications of the corresponding ones in the parent. Thus FS includes formal taxonomic relations such as that between animal and mammal, while going further to include the concept that one computer further specifies some known computer configuration in having extra memory.

- Descriptive. This category includes various attributes that don't fit elsewhere, eg, cycle time, memory size, and bandwidth. Further classification at some later date is not precluded, it is hoped.

- Physical. This includes: spatial layout constraints, cabinet size, power supply requirements, cooling requirements, and noise, to name a few.

- Functional. Here are collected properties having to do with how a component functions or behaves within an assembly of other components. Specifications of inputs and outputs are examples. Moore's work on MERLIN processes [15] and that of Freeman and Newell [9] are good starting points for this area.

- Means-ends. Primarily used for methods in IPMSL at the present time, the relations here directly correspond to the language for expressing methods, to be discussed in detail below.

Figure 1 depicts a fragment of a network including three of the subnets (the structural subnet is represented graphically; this picture is manually generated). The computer that is partially depicted there is the DEC VAX-11/780, a recent medium-scale computer.

[2] I don't know how this use of WM corresponds to dynamic state memory in conventional semantic net systems (due to ignorance of their implementation, which is rarely, if ever, discussed in the literature), but it appears that this use of WM provides a great deal of flexibility and breadth of focus, allowing processing of the network information in many useful ways.

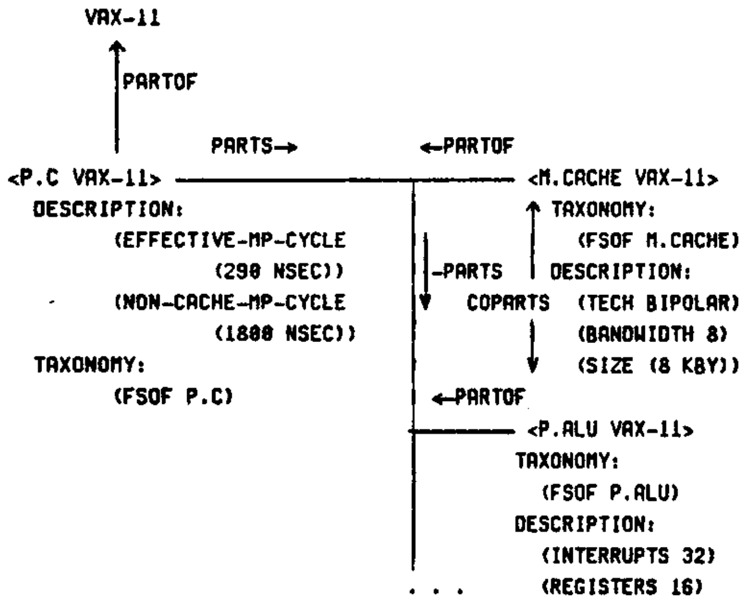

Figure 1: A fragment of a PMS network

The figure suppresses some details about the actual node and link structure; it is intended primarily to be an example of the kinds of information dealt with, and of how the six subnets partition them. Note that the division into subnets essentially assigns one set of nodes and interconnects them using six different sets of relations. (The subnets are not distinct areas of the net, as may be the usage elsewhere in the semantic network literature.) As will become evident from details below, the implementation actually attaches six subnet nodes to each main node, one for each subnet, and then attaches relations emanating from that node to the appropriate subnet nodes. Relations coming into a node from others in the net all point at the one main node, however (ie, they use only the name of the node).

Why have the network relations partitioned into these six subnets?

- There are problems that are solvable using only one or two of the subnets. Thus the subnets organize the knowledge modularly according to how it will be accessed. However, there are problems that are expected to integrate information from all of the subnets. Compare a similar situation in engineering a building: heating, plumbing, and electrical wiring can be considered independently for the most part, but the finished design will have worked out the few interactions and exploited common structural conduits.

- Methods for processing the network are specialized in a natural way to work with each subnet. The subnets demand such specialization due to different inferences (ie, logical properties) associated with each.

- It aids the production system implementation, in allowing separate productions to store corresponding subnet information. When data is evoked into WM, the separation allows it to be evoked selectively, thus economizing on WM space. This is a kind of automatic restriction of network search.

- There are precedents among traditional semantic network implementations but sufficient detail is usually not given to display how they actually profit from or exploit such a division. E.g., Hayes [10] distinguishes between taxonomic and structural links in a similar way. Generally little attention is given to this issue, particularly from the standpoint of how much the decision to divide might be forced by the domain content rather than by the underlying language architecture (in the present case, there is in fact good reason to do it on production systems grounds alone).

- The structuring is a conceptual aid to the user and designer of networks and of methods to work on them. It clarifies the overall structure of the knowledge, and forces knowledge coming into the net into a particular mold. This is extra knowledge content, in a sense, that IPMSL can exploit in treating the network and in solving problems. (If there is psychological validity to the production system model of human memory, such a division makes the information easier for humans to process in the same way that it is easier for IPMSL, particularly in economizing WM capacity.)

- Knowledge can be added to the net unevenly, with different subnets receiving emphasis at different times. Since method organization (modularization) corresponds to the division, incomplete information, when isolated to a particular subnet, need not interfere with processing elsewhere.

Now for some details on the representation. First is given a picture of how the information looks dynamically, in WM. Then productions are discussed.

Nodes in WM are temporary and consist of temporary symbols with attribute-value pairs attached. There is a main node for each object in the network, and a subnet node for each subnet that is defined and evoked. Figure 2 shows the WM elements for a typical component, with Lisp internal formats translated to a more readable graphic arrangement.

```
S0123                    S0136                      S0138
  KNOWLEDGE-ABOUT P.C      KNOWLEDGE-ABOUT P.C        KNOWLEDGE-ABOUT P.C
  ROLE       MAIN          NET      S0123             NET     S0123
  STRUCTURE  S0136         SUBNET STRUCTURE           SUBNET TAXONOMY
  TAXONOMY   S0138         PARTS    (D.ALU M.REG)     FSOF    P
                           PARTOF C                   FSS     (<P.C LSI-11> <P.C VAX-11>)
                           STRUCTURE P109             TAXONOMY P202
```

Figure 2: WM elements for a typical component

Figure 2 includes a main node and two subnet nodes for the P.c (central processor) component: S0123 is the temporary name for the main node, S0136, for the structure subnet node, and S0138, for the taxonomy subnet node. Each node is identified by the "knowledge-about" attribute - this serves to name and to tie together the three nodes (all of which are "knowledge-about" P.c), in such a way that pattern- matching in rules can easily collect together the separate nodes. The main node has two pointers to its subnet nodes, and each subnet node has a pointer back to the main node. The two subnet nodes include, as values of "structure" and "taxonomy" attributes, the names of the productions that built them, so that long-term modifications can be done. (The main node is built by a general production, so it doesn't need such a pointer.)

The production that builds the subnet node for the structure of P.c is the following:

```
P190: IF THE GOAL IS KNOWLEDGE-ABOUT THE STRUCTURE SUBNET OF P.C
   AND THERE IS KNOWLEDGE-ABOUT THE NODE OF P.C WITH THE MAIN ROLE,
THEN BUILD A KNOWLEDGE-ABOUT P.C SUBNODE WITH SUBNET STRUCTURE, WITH
      NET THE NODE OF P.C WITH THE MAIN ROLE, WITH PARTS (D.ALU M.REG),
      AND WITH PARTOF C,
   AND MARK THE GOAL SATISFIED
   AND ADD TO THE NODE OF P.C WITH THE MAIN ROLE THAT STRUCTURE IS
      THE NEW SUBNODE.
```

The syntax used here is a hand translation from the list-structure version, for readability. Some clumsiness remains, due to the effort to retain as much of the attribute-value flavor of the representation as possible. "Of" and "with" denote a relation between a pair and the object name to which the attributes apply.

All such network-information rules have a similar structure: a simple "if" part that detects the goal to access the knowledge and gets from WM existing information about the main node for the component desired; and a "then" part that constructs the appropriate dynamic node for the particular subnet, along with updating pointers back and forth. The rule P190 is specific to P.c, with other objects stored in the net analogously, each with a specific set of rules (one for each subnet).

Two general productions are involved in accessing network knowledge, one to build the "main role" node when it is absent from WM, and the other to notice that the knowledge desired is already in WM, and thus to mark the "knowledge-about" goal to be satisfied. Being general means that they apply to all objects stored in the net. These are P98 and P99, respectively, and are displayed below:

```
P98: IF THE GOAL IS KNOWLEDGE-ABOUT SOME SUBNET OF SOME OBJECT
   AND THERE IS NO KNOWLEDGE-ABOUT THE NODE OF THAT OBJECT WITH
      THE MAIN ROLE,
THEN BUILD A KNOWLEDGE-ABOUT NODE FOR THAT OBJECT WITH ROLE MAIN.

P99: IF THE GOAL IS KNOWLEDGE-ABOUT SOME SUBNET OF SOME OBJECT
   AND THERE IS KNOWLEDGE-ABOUT THE NODE OF THAT OBJECT WITH
   THE MAIN ROLE WITH THAT SUBNET BEING SOME SUBNODE
   AND THERE IS KNOWLEDGE-ABOUT THAT SUBNODE,
THEN MARK THE GOAL SATISFIED.
```

Note that "some" and "that" refer to unrestricted pattern_variables in the actual productions; the word following them is not a semantic restriction on values assumed by the variables, but enhances human readability only. The way that these three types of knowledge-access productions work together is dependent on the conflict resolution strategy of OPS2, particularly the special-case criterion, by which the rule that matches more data is preferred. P98 is the least special-case of the three, and its condition is explicit enough that it will be true only when the others cannot be, so special-case is not so critical for it. But neither P190 (and all others of its type) nor P99 has any negative conditions, so P190 can be true at the same time as P99, and here the special-case criterion makes the difference.

One other form of access is used: access to a particular attribute of an object, where the particular subnet is not known, an example of which is P145:

```
P145: IF THE GOAL IS KNOWLEDGE-ABOUT THE PARTS ATTRIBUTE OF SOME OBJECT,
   THEN ADD TO THE GOAL THAT THE SUBNET IS STRUCTURE.
```

There is a production like P145 for each attribute that is known to the system. Note that P145 remains true even after it fires, but that conflict resolution prevents it from repeating in a couple of ways (one is sufficient, of course): the refraction principle inhibits such repetition explicitly; lexicographic recency will give preference to other rules whose conditions include the subnet information that P145 adds - the productions displayed above are all good candidates, so that those others will fire next.

The kind of action done by P145 is typical of the production system style used here: a goal is a symbol structure to which a variety of rules can contribute small pieces, until enough is accumulated to make the goal satisfied. It should also be noted that the ability to flexibly accumulate such data depends on the attribute-value set representation used in OPS3RX.

2.3. Basic IPMSL methods

The above has sketched the mechanisms of basic access of information in IPMSL. Now we turn to three central IPMSL commands, which build on the basic access to form actions with wider effects.[1]

A user causes the construction of new network nodes with the *DEFINE* command. The method for doing this involves taking the information from the user and embedding it in the fixed structure common to all knowledge-access productions, of which P190 above is an example. How one specifies productions to be built is the subject of section 3. An example of a define command follows (the "*" is the system's prompt character). This defines part of Figure 1.

```
*define structure (name (P.c VAX-11)) (partof VAX-11) ,
*    (parts ((M.cache vax-11) (P.alu VAX-11) ::))
```

Modification of existing net structures is done with the *LET* command, which includes the attribute and the new value of the symbol to be modified. WM structures are modified directly. PM structures are accessed via the appropriate pointers in WM (ie, rule names that are kept with each node) and then their actions are edited so that future executions will result in the modified structures being built in WM. Usually the Let command has the proper subnet filled in by rules that know which attributes are part of which subnets, eg, P145 above. This subnet inferral is done in the process of accessing the knowledge, as part of Let. The purpose of this access is to check that the Let command is actually specifying a change to existing information. The Let command below says, "let the FSof property of P.c be P".

```
*let FSof P.c P
```

As a result of Let, there is often a need to update other pointers so that they are consistent with the new information. For instance, if a Partof connection is added, a change must be made to the corresponding converse relation, Parts. Let constructs a subgoal to check whether there is a converse to be updated, and that subgoal is processed by rules similar to the following:

[1]The command examples given here are exactly as they appear for the basic IPMSL system. Subsequent changes as IPMSL has grown have smoothed out the syntax of interaction somewhat, though making commands significantly more natural-language-like has been postponed.

```
P170: IF THE GOAL IS TO UPDATE THE CONVERSE OF THE ATTRIBUTE PARTOF
        IN A NODE WITH SUBNET STRUCTURE AND WITH SYMBOL SOME S
        AND WITH VALUE SOME V
THEN SET UP A NEW GOAL TO LET THE STRUCTURE OF THE NODE WITH SYMBOL
        THAT V HAVE ATTRIBUTE PARTS AND VALUE THAT S
     AND MARK THE FIRST GOAL SATISFIED.
```

Note that this amounts to a new Let subgoal being generated as a result of a higher-level Let goal, illustrating that such actions can be both user-initiated and internally goal-initiated. In fact, in some cases Let can lead to the construction of new Define goals, when there needs to be created a new node in the network to hold some information resulting from converse updating operations.

The user can evoke a region of the network and have it displayed in a tree format by using the *SHOW* command, as in Figure 3. (The tree is sideways, with indentations indicating parent-daughter relations.) This command sets up goals to search from a starting node through all nodes related to it in specified ways. At each node visited, information is collected into a tree structure, and after the search is completed, this tree is printed.

The Show command has other options, eg, a "with" option, which specifies what kind of additional information (eg, the data stored in the description subnode) is to be collected at each node, for display.[1]

2.4. Discussion

To summarize, the basic IPMSL commands provide access (query), definition, modification, and search-and-display capabilities. Given additional appropriate domain information, namely basic knowledge about computer structures (their taxonomic, spatial, structural, functional, etc. properties), the system is ready to attack the configuration problem described above.

How general is this basic portion of IPMSL? That is, can it be applied readily to other domains? Yes, for the following reasons.

- Define and Let are coded to be general. They don't incorporate knowledge about specific subnets (taxonomy, et al), so that the user can divide up his domain knowledge along other lines. The use of subnets, however, and the accompanying node structures, is built in. The user of new subnets can provide information about them in the form of simple rules such as P145 (saying which subnet an attribute belongs to) and P170 (for converse relation updating).

[1]Several features have been added to Show during the growth beyond the basic IPMSL, eg, more graphical output and new kinds of searches for useful extra information.

```
*show taxonomy pms.top

*****
PMS.TOP D <D.CLOCKS VAX-11>
        T <T.DIAG.REMOTE VAX-11>
          <T.CONSOLE VAX-11>
        L <L.UB VAX-11>
          <L.MB VAX-11>
          <L.SBI VAX-11>
        H <H.DIAG VAX-11>
          <H.SBI VAX-11>
        K <K.PC-SBI VAX-11>
          <K.PC-DIAG VAX-11>
          <K.FLOPPY VAX-11>
          <K.CONSOLE VAX-11>
          <K.MP VAX-11>
          <K.UBA VAX-11>
          <K.MBA VAX-11>
          K.IO
        P P.ALU <P.ALU VAX-11>
          P.C <P.C LSI-11> <P.DIAG VAX-11>
              <P.C VAX-11>
        M M.MICROCODE <M.DIAG VAX-11>
          M.CACHE <M.CACHE VAX-11>
          M.BUFFER <M.ADRCACHE VAX-11>
                   <M.INSTR VAX-11>
          M.S <M.FLOPPY VAX-11>
          M.P <M.P VAX-11>
        C VAX-11
*****
```

Figure 3: A display of part of the network

- It is easy to augment the existing Show methods to work with new net structures. Eg, the "with" option is already somewhat general. Further generality might be added, and other features can be added in by analogy with existing rules.

- Throughout the basic commands, there is included no assumption about anything related to the PMS domain. IPMSL is thus a basic semantic network engine that can be applied to problems in a number of domains (but see Sections 4 and 5).

What have been the major implementation and design issues for IPMSL?

- How to use the production system architecture? Besides the alternative chosen and discussed above, there are other ways. The most important one involves not using rules to store knowledge, but having another separate memory for the net, and allowing both patterns and actions to test and have effects on that memory, as in Anderson's ACT system [1]. The present approach is preferred primarily because it makes more contact with techniques and software developed already for "pure" production systems, ie, it doesn't make significant additions to the existing and well-understood architecture. Thus we can proceed immediately to domain content issues. Some aspects of Anderson's system seem inappropriate for our domain, and some issues in his system are not central to ours.

- How does this use of the PSA compare with others that are equally straightforward, eg, having a rule represent each attribute-value pair associated with a node, which in some cases amounts to having the "arrow" in the rule correspond directly with a semantic network arc? In a sense, the approach here is more controlled, ie, there is less chance that information will develop from given information in an unguided way - goals are required in all cases to evoke new facts. One can, however, set up a method that results in such unguided evocations, because the needed goals are expressed uniformly. In addition, more control is added to the knowledge structure due to the partitioning of facts into subnets. In another sense, the approach here is less controlled: one gets all the facts in a subnet at once, rather than evoking a single precisely-requested fact. Thus at any time one can obtain unexpected data items that could, for instance, satisfy any one of a number of pending goals. The present position, thus, strikes a balance between strict control and absence of control.

- The decision to partition the net into subnets is important, for the reasons stated above.

- Some issues with respect to taxonomy, naming, contexts, the six subnets chosen, etc. are discussed in Section 5.

- Efficiency is a concern, since IPMSL is intended to be worked with interactively. Response times for network definition commands are in the range of one to two minutes on our time-shared DEC KL-10. This usually is significantly exceeded by the time it takes a user to decide how to formulate the next piece of network to be added, and is therefore a tolerable response delay. Having the system run interactively seems critical due to the complex effects that can occur with each addition or change to the network.

- The use of *symbolization* is a key issue in IPMSL. Nodes in the network are arbitrary symbols (GENSYMs) to which names are attached (via the "knowledge-about" attribute) along with other attribute-value pairs, rather than being sets of pairs attached to the name itself. Another occurrence of the same principle is in having separate symbolized nodes for the six subnets. Goals and other data structures in IPMSL, due to their being symbolized, can flexibly accumulate records of changes in state and growth in basic facts relevant to their solution. The network itself is directly amenable to dynamic re-organization in WM, by hooking together symbols in new ways. This easily gives, for instance, dynamic "frames", "depictions" [10], and other higher-level structures. Symbols are important at a low level in this implementation because of the permanent-temporary dichotomy induced by having PM and WM. When one transfers information from WM to PM, as in building up new parts of the net, it is very clumsy to have to deal with a named structure in WM of which only part of its attribute-value pairs are right for transfer. PM reflects the same distinction in not containing the actual symbols or any other arbitrary names, but simply pattern variables - symbol names are either matched to the variables during recognition or are created anew during action execution, and it is all the same to the system. In a sense, then, symbolization is ubiquitous while being so natural a part of the rules that symbols become invisible to a reader or coder - an important advantage to using rules.

3. The IPMSL approach to instruction and augmentation

IPMSL started out as a small set of productions (less than 50) called Kernl2 (Kernel, version 2). Kernl2 allows simple kinds of interaction to take place, including interaction that leads to adding new rules. The first interactions with Kernl2 involved adding to and improving Kernl2's capabilities, while later ones were increasingly devoted to adding the basic IPMSL commands. Kernl2's essence is a set of methods for interpreting and executing user inputs that are expressions in a simple method language. This method language allows the user to designate elementary components of a method. After a number of such designations, the user gives a command essentially saying that the method is ready to be formed into productions, and Kernl2 does the rest of the work: keeping track of what is designated and finally putting it together into rules. Kernl2 also includes rules that describe itself in a declarative way, ie, there is an IPMSL-style network describing Kernl2 methods. Thus, a user can both construct methods of his own for new goals and access the Kernl2 network to understand and augment Kernl2 itself. Both of these operations have in fact taken place as IPMSL has grown. The existence of the self-describing network in Kernl2 aims at the possibility that techniques (to be developed) for describing and manipulating external systems (eg, computer structures) will ultimately find application in the understanding by the system of itself.

The approach to growing a system, ie to instruction, that is presented here builds on two previous independent approaches: that of the Instructable Production System project [20] and that of Waterman's Exemplary Programming [26]. It is beyond the scope of this paper to give a detailed comparison to these approaches or to others for augmenting rule-based systems.

The idea behind having Kernl2 be a part of what is basically a semantic network system has three aspects. First, the method language of Kernl2 allows a user to easily build up new system behavior, ie, it is a means to making the system fully extensible. In addition, Kernl2's self-describing "help" network makes it possible for a user to change the kernel itself. Second, the nature of the method language and of the way it is used is designed to provide a better way of coding and debugging the rules, thus maximizing as much as possible the speed of system growth. The critical aspect of the language that allows this is that method construction takes place within a dynamic WM context that is similar to that in which the method will work after it is finished. Thus in a sense a user is hand-simulating the method's operation while building it up, so that by examining WM the user can easily note those items of information that the method should take into account and also determine in a natural way those actions that the method should perform plus their proper sequence. This will be explained more in the following. Third, all of the method language facilities are available both to the user and to internal methods built from rules. That is, they are all considered goals internally, and can be set up by other rules and asserted into WM in the usual recognize-act cycle in order to obtain the desired effects. In particular, one can compose methods that go through the various steps to build up further methods - the Define command of IPMSL does precisely this. Note that the method language itself is thus extensible.

The method language of Kernl2 consists of four main commands plus several other operations. As a collection, these constitute the way that rules are added. (Other parts of Kernl2 are discussed below.) The four main commands are as follows.

GOAL. This specifies that some symbol in WM, given as an argument to the command, is the main goal for a new method, or the goal for a new rule within a method *already partially* completed. Goals are the primary control-organizing feature in IPMSL methods: they state what is to be accomplished, record progress towards that desired state, and finally also *record* the outcome. A goal is an ordinary WM element in most respects, the exception being a special marker that specifies its "goal" nature. It has a number of attribute-value pairs, and the various functions that a goal serves are represented as an accumulation and modification of the goal's set of pairs.

The following shows a new goal being given to the system, with the intention next to start to instruct it on how to achieve it. Throughout the rest of this section, lines starting with "*" are inputs from the user to the system. The fragments of input and output from the system given here are not a continuous stream of everything done, but rather a selection of the most relevant information.

```
*tell something (about c)

S0024 (ABOUT C)
      (%MF WI-- TELL SOMETHING)
      (MS INTERPRET)
```

This goal (created for this demonstration, but similar in spirit to goals that the system has been taught) means that IPMSL is to find something in WM regarding "C" to tell the user - the system already has goals to tell specific things, but this when completed would have IPMSL pick something of interest to say. "S0024" is an internal symbol created to hold information from the user's input. The user's input is cast into attribute-value form partially automatically. In this case, the first two input words become part of the "%MF" attribute of the goal - %MF being a "header" item marker (its weird form is dictated by efficiency considerations). The symbol is marked as a goal by giving it the "WI--" tag within the %MF attribute - "WI" standing for "want initiate". The pair "(about c)" is carried over into S0024 directly from the input, and the "(MS interpret)" is added to S0024 to indicated that its source is the "interpret" method, as opposed to, say, a goal from an internal system source. I have omitted an output by IPMSL stating that this "tell something" goal is new to it.

Next the user points to S0024 as being a goal for a method to be instructed.

```
*goal s0024

S0050 (P P333)
      (ANTECEDENT S0024)
      (SYMBOL S0024)
      (%MF H--T GOAL II)
      (MS INTERPRET)
```

From network information stored about the "Goal" command, the interpret method has decided that S0024 is the "symbol" attribute of a new goal. S0050 is a data structure that will collect information about the new method being instructed. The method for performing the Goal command has already "initialized" S0050 with the facts that a new production for a new method will be P333, that P333 will have S0024 as an antecedent (condition), that S0050 is what I have been saying it is (the %MF header data), and that S0050 also came from the interpret method.

USE. This specifies that some WM symbol is to be taken as a condition in the rule currently being built up. In the following, the first command sets up some instruction context, so that the "Use" can point to something meaningful.

```
*c

S0066 (FUNCTION P117)
      (%MF H-K- C II)
      (SUBNET FUNCTION)
      (NET S0065)
      (OUTPUTS I-UNITS)
      (INPUTS I-UNITS)

*use s0066
```

Primarily, this Use command results in adding to S0050 above the new fact that S0066 is also to be an antecedent of P333, the new production. S0066 is a fragment of the network describing the "function" aspects of the PMS component "C" (computer). The interpreter assumes that one wants functional information about something, when nothing but its name is given. That "C" is used here is only for instructive purposes. A little later, a command will be given to cause it to be generalized to any component, and the "function" subnet will also be generalized - thus "tell something" will ultimately pick up any network fragment for any component.

The instructor now proceeds to tell the system what it is going to do with the WM information picked up by the pattern conditions in P333.

ADD. This specifies that a symbol is to be taken as an action in the rule being built up. The following first shows a command by which the instructor gets a suitable goal into WM for such an action, and then shows the corresponding add command.

```
*show object s0066

S0103 (%MF OI-- SHOW OBJECT)
      (OBJECT S0066)
      (MS INTERPRET)

*add s0103
```

The instructor asks the system to show the object S0066, expecting to have the same action incorporated into the new production. Since this is a known goal, it is actually executed and marked old - its completion is marked with the "OI--" in the %MF header information. This will have to be changed later, back to "WI--", so that when the production fires, the goal that it produces will be an active one. The Add command puts another datum into the set of attribute-value pairs for S0050, stating that S0103 will be a consequent (action) in P333.

Except for a few finishing touches, the data is set up for the new production. The commands shown next do a few odd things to the data. These commands are basically WM editing commands, whose effects will soon be transferred to PM because they alter parts of rules being formed.

```
*notice attribute c bvar p333
*notice attribute function bvar p333
*strip attributes s0066 (attributes (function net outputs inputs ::))
*strip attributes s0024 (attributes (ms ::))
*strip attributes s0103 (attributes (ms ::))
```

The two "notice" commands specify that "c" and "function" are going to be generalized into arbitrary variables in P333 - BVAR is a special WM symbol whose attributes are names to be made into variables and whose values are the names of the productions in which that is to be done. The "notice" commands thus cause the right attribute-value pairs to be added to BVAR. The "strip" commands cause the named attributes to be removed from the named symbols - the odd syntax is a result of the primitiveness of the interpreter. The results of these commands will be evident in the rule to be displayed below.

BUILD. This takes the pieces of a rule that have been specified with the other commands, and forms them into a rule. The rule then becomes part of PM. The target rule of the above dialog might be translated as,

```
IF THE GOAL IS TO TELL SOMETHING ABOUT SOME OBJECT
   AND THERE IS KNOWLEDGE-ABOUT THAT OBJECT WITH SUBNET SOME X,
THEN ASSERT A NEW GOAL TO SHOW THE OBJECT THAT IS THE KNOWLEDGE-ABOUT
      THAT OBJECT.
```

Here is the appropriate Build:

```
*build p333

P333 ( (=B802 (%MF WI-- TELL SOMETHING) (ABOUT =B805))
    (=B801 (%MF H-K- =B805 II) (SUBNET =B804))
--> (=B803 (%MF WI-- SHOW OBJECT) (OBJECT =B801)) )
```

The rule P333 is shown after a manual edit was done to change the status tag in the action from "OI--" to "WI--", as explained above (the method language does not yet provide a better way to do this). Note that the symbols S0024, S0066, and S0103 have been automatically generalized into pattern variables (prefixed by "="), and that the other variables have been created as specified in the BVAR object.

While the preceding dialog with the instruction-taking part of IPMSL may not show the full power of this mode of augmenting the system, and while its advantages over directly entering the production are not clear due to the simplicity of the example, the dialog does show some of the flavor of how instruction can take place in a dynamic context. Along the way, the instructor has been shown accessing the network and evoking subgoals, using the results of those commands as part of the new rule.

In addition to the four main commands just sketched, there are a number of auxiliary commands. One class of auxiliary has to do with modifying WM contents so that the rule being built has the right form; this class includes ways to add and modify the attributes of existing WM elements and ways to assert new WM elements. A second class of auxiliary uses the dynamic context in WM more actively, by evoking subgoals, for which methods already exist. The effects of such subgoals are observed, and their results noted for possible inclusion in the rule being built. A third class involves setting up data structures in WM that will bring about modifications in the way that the rule-building function (evoked by the Build command) works. For example, one can specify that some constant in the WM elements that are included should be converted into a pattern variable (a form of instant generalization). A fourth class of auxiliary[1] can be termed utility functions: they are useful for displaying the current state of WM, including the effects of previous commands. A particularly useful utility is the Hold function, by which the user can keep track of, and periodically display, a particular list of symbols - usually just those that the user has designated to be part of a new rule. A fifth class of command allows the user to edit existing rules, in practice an important part of each instruction session. This is greatly aided by the network of rules that describes the methods in the system itself. The network and the method language are coordinated to some extent: when the language is used to change a method, the corresponding descriptive network is in many cases changed along with it.

Figure 4 is a summary of all components of IPMSL, with Kernl2 methods listed first.[2] The scope of the PMS and VAX networks (95 rules) is as follows: basic taxonomic information about the abstract single-letter components (described at the beginning of the paper); taxonomic information about other abstract components, eg, P.c (central processor) and M.p (primary memory); structural, taxonomic, and descriptive information about the VAX-11, except for peripheral I/O devices (these would ordinarily be given during specification of various saleable configurations).

Kernl2's methods, displayed in the table, can be divided into several classes: taking in user sentences (input, interpret, help), displaying information (tell, show), main method language methods (goal, use, add, build) and auxiliary method language methods (edit, notice, strip). Kernl2 started out at about half its present size, with the additions done mostly using the method language. By most standards, Kernl2 is quite a small program, but has nevertheless proven to be sufficient for starting out interactively to build a much bigger system (which is still in progress).

[1]The fourth and fifth classes are not rule methods, but Lisp code.

[2]Subsequent growth has occurred in the nets and methods, as detailed at the end of the paper, beyond this basic IPMSL.

Method	Number of rules	
Input Sentence	2	
Interpret Sentence	19	
Help User	8	
Convert Help	8	
Clear display	4	
Kernl2 help net	21	
Tell Object	7	
Show Object	1	
Goal	1	
Use	2	
Add	2	
Build	1	
Build Schema	2	
Edit	1	
Notice Attribute	11	
Strip Attribute	9	
(total Kernl2)		99 (31%)
Define	19	
Let	33	
Union lists	9	
Update Converse	12	
Find knowledge	18	
Show (subnet)	7	
Tree (subnet)	24	
(total IPMSL basic)		122 (39%)
PMS (abstract)	23	
VAX-11	71	
LSI-11	1	
(total PMS nets)		95 (30%)
(total IPMSL system)	316	

Figure 4: Summary of IPMSL methods

While the Kernl2 method language has proven effective conceptually, it is less effective in practice than a more direct approach of simply composing rules non-interactively and entering them into the system. The "less effective" refers only to time efficiency or productivity of an expert rule coder (the author) - the response time of our local time-sharing system (TOPS-10 on a DEC KL-10) running IPMSL is so slow (on the order of a minute or two per user input) that it is not effective to use recognize-act cycles for this when rules can be composed and entered directly. If run-time efficiency could be improved by an order of magnitude, say up to about 20 to 50 rule firings per second of machine run time, which would reduce response time to less than 20 seconds under normal load conditions, or if IPMSL could be run on a dedicated or lightly loaded computer, then the balance would shift back to using the method language and building the system fully interactively. Another part of the present shift away from the method language is due to a change in overall goals to an emphasis on obtaining rapidly a fairly large system (on the order of 1000 rules). Thus unfortunately many interesting AI issues surrounding instruction and the use of the method language in more complex situations have been postponed. Even so, the method language might be effective and useful for a naive user, one who is not familiar with the system, and for a user who cannot code rules easily in the non-interactive manner.

As a result of this strategy shift, some auxiliary software has been added to aid the direct-coding mode of user interaction. This includes a facility for abstracting and displaying all or a select part of the rules in a particular method in a relatively small screen space; more flexible rule entry and editing, including the ability to copy information from one rule into another (thus one can make a new rule "like" another, and then edit in the discriminating elements); and the ability to make listings of the rules organized according to method. With these aids, it is possible to grow and debug the system at a rate of about two to five rules per on-line hour.[1]

4. Advantages and disadvantages of using rules for a network

The evaluation of production rules as a basis for a semantic network system consists so far of a set of subjective impressions and design characteristics. The existing implementation of a net is an unusual mix of procedural and declarative components. The net is active in a real sense, though controlled by particular activation goals. Control in general for the net can be distributed around, as rules, but since WM is global and inspectable at all cycles by all rules, there can be global (centralized) control to a large extent. As yet, there has been no problem of searches in the net getting out of control, and in particular, unexpected rule firings have not interfered with processing.

The following lists a number of specific advantages that the PSA seems to have for network systems. For the most part, since the scope of IPMSL's application has been limited, these should not be taken as soundly demonstrated as yet.

- WM serves as a large dynamic context. It records uniformly both the state of methods that are being executed and the state of the network as it is searched. Searches and methods both respond to the same sort of event.[2]

- The single shared dynamic state allows flexible control of searches. That is,

[1]Historically, the system has grown at a rate of about 80 to 100 rules per month.

[2]I don't know whether conventional net implementations have a structure that corresponds in function or in flexibility to this WM.

ordinary searches can be monitored by global, general rules, by specific search heuristics, and by control knowledge that is distributed throughout the net - all expressed as rules.[1]

- WM is useful for growing large, hybrid, temporary structures and mappings - things that one would not necessarily want to become a permanent part of the net.

- Rules can bring together (by recognition) complex patterns of diverse knowledge, thus making it possible to integrate information in new ways.

- The rules are readily organized into an instructable structure. Growth is by accumulating more rules, each of which is a readily constructed and comprehended unit. Rather than procedural attachment, there is a more intimate co-existence of procedures (methods) and data (network rules).

- Searching methods and others can themselves be described using the same network conventions as the subject domain. Methods are structurally quite simple, as is induced by the method language, consisting primarily of simple goal-subgoal dependencies and simple functional (input) requirements.

- The organization and control of methods and search by goal-subgoal structures can, in addition to usual forms of control, offer a form of backtracking. Control can "fall back" to an unsatisfied goal and then go forward in a new direction in a further attempt to satisfy it. This control is enabled by the OPS2 conflict resolution strategies. Goals are interrelated in ways that can be much more open (heterarchical) than conventional recursive (hierarchical) forms: the goals are global structures in WM that can be processed in varied orders, can be satisfied in accidental ways, and can be examined and re-ordered flexibly.

- Existing efficiency techniques for production-rule systems can be immediately carried over to network searches. In fact, the rules in OPS2 are compiled by converting their patterns into a very efficient network structure, developed by Forgy [8]. Efficiency has been factored off and studied independently as a problem in its own right, and the results of efficiency improvements in that general pattern-matching domain can now apply to networks of rules. The ability to exploit the available power is limited somewhat by the simplicity of patterns in existing net rules, and by the inherently serial nature of much of the network search when implemented as rules.

- The recognition part of the execution cycle is amenable to simple parallel implementation. Thus there is a simple way for enormous databases to map onto one of a wide variety of parallel processors, composed of large numbers of small, cheap microprocessors (work on the details of this is in progress).

- The net need not be uniformly encoded: one could do various arbitrary things, using rules, to evoke information (or to respond to it) in special cases.

[1] I doubt that a network could match this flexibility without using rules or rule-like entities (eg, "demons").

The above positive features can be balanced with the following disadvantages.

- Space usage seems high: a few attribute-value pairs require a good bit of surrounding rule structure.[1] This is offset by the fact that rules are fairly fixed structures, so that they are amenable to compactification and more efficient storage techniques than are arbitrary list structures. Also, the Forgy RETE network [8] causes patterns to share storage for identical subparts to a great extent.

- Search is serial: nodes are developed one at a time. But as mentioned above, given a parallel processing architecture, production systems are just as favorable for exploiting it as are conventional network organizations, if not more so. Other possibilities for alleviating this will be discussed further below.

5. Traditional semantic net issues

The discussion of IPMSL so far has not touched on a number of issues and capabilities that have been traditionally connected with semantic networks.[2] This section will touch on a few such topics, with the intention of determining whether such issues will present the IPMSL approach with insurmountable difficulties. In cases where there do appear to be serious difficulties, it can be argued that the IPMSL task domain is such that the difficulties will not impair IPMSL's performance or potential. That is, the task of describing and manipulating computer descriptions does not make the same demands as other semantic net tasks, and demands that might provide difficulties for IPMSL do not occur in it.

5.1. What kind of network is IPMSL?

A major distinction made in the literature involves the term "semantic network" itself. Early semantic networks (eg, Quillian's) did in fact deal with semantics: they were networks of dictionary-like word meanings, interconnected by meaning-association links. Other networks have encoded entirely different sorts of knowledge: knowledge about events, causes and effects, people's motivations with respect to events, etc. Examples are the networks of the Norman-Lindsay-Rumelhart group, Schank, and Simmons (see [4] for a bibliography). Following Schank's usage of "episodic", I call these episodic networks. Each of these two classes of network is concerned with different phenomena and encounters different problems. In which category does IPMSL fall? The semantic-episodic distinction

[1]I don't know how this compares to conventional implementations, but they certainly also must use some extra storage structure.

[2]Some of these, while drawn from a number of sources, have been discussed and clarified recently by Fahlman [6]

seems to be based on the philosophical distinction between analytical and empirical facts (this and other distinctions to follow are in part based on Sandewall's discussion [21]). The former are facts that are determined by purely definitional properties of components, while the latter deal with matters of observation in a real environment.[1] Semantic nets, then, are analytical, while episodic ones are largely empirical. IPMSL combines both abstract definitional knowledge about computer structures and knowledge about particular computers, though in the latter category, the knowledge is still largely definitional (eg, from hardware specifications rather than from observing one in operation). Thus IPMSL is a semantic network. But it is easy to imagine a system in the same class as IPMSL, ie, concerned with symbolic description and manipulation of systems, where empirical knowledge would be important, eg, a system for describing a city. It is possible that IPMSL, in its advanced form, will use empirical data or at least test its conclusions against it (eg, in the case of simulating a subject system's behavior).

Another dichotomy that might serve to distinguish semantic from episodic networks is that between the description of events and the description of states or objects. Episodic nets describe primarily events, while semantic nets describe objects. Schank's approach, for example, contains little or no mechanism for describing objects (eg, it is not a good system in which to describe such facts as most humans have two arms, each arm has a hand, each hand, five fingers, etc.). IPMSL's status under this dichotomy is still as a semantic net, but events (eg, data and control flows along links) are expected to be an important part of the eventual system, so there is some ambiguity.

Semantic nets can also be distinguished in containing abstract (synonyms: generic, general) kinds of facts, while episodic nets are mainly concerned with specific, concrete ones. IPMSL spans both kinds of knowledge in this dichotomy. The FSof (further specification) relation glides across the boundary without making any kind of fundamental change in terminology. Both abstract and concrete objects can be further specified, and it can be unclear when or whether the boundary between abstract and concrete is crossed. IPMSL shares this trait with MERLIN [15, 16]. The famous type-token distinction has no place in IPMSL for this reason. A token is usually considered a unique instance or individual of a more abstract "type" object; it is generally illegal to make further tokens of a token. But IPMSL has no such "ground" symbols: everything can have further specifications (discussed further below). This infinite FSof-chaining ability is demanded by the PMS domain, and probably by all domains in the same class.

This discussion has, among other things, established a major distinction in classes of

[1]Woods [28] has a similar distinction: defining versus asserting properties.

network, in order that further discussion of IPMSL issues can restrict itself to those of the *semantic network* class and can ignore issues and phenomena of concern to builders of episodic networks. Thus, for instance, it suffices to discuss issues of inheritance and other taxonomic phenomena, to the exclusion of issues of what case slots or action primitives to use. It suffices to consider searches along semantic links and spreading activation, to the exclusion of indexing of events to allow retrieval based on memory cues of partial graph patterns.

This paper further restricts itself to basic issues of implementation and to those issues that bear on the novelties of the present system's approach. As an implementation of a semantic network with certain design features, this system will have representation problems in common with a number of others, with respect to contents of what goes into the net. This paper will not concern itself with them unless there appears to be some reason to believe that solutions to them might interact with the implementation as rules. Some typical problems in this class are mass terms and non-binary relations. The assumption here is that production-rules are as open and as flexible to respond to substantive problems and to changes in contents' structure as are other possible implementations.

5.2. Taxonomy

A number of approaches have been used in semantic networks to express taxonomic relations: ISA, superset-subset, superconcept, generalization, specialization, set member, instantiation, individuation, type-token, generic-specific. Of these, ISA is probably the most offensive, unless some care is taken in defining it. It can suffer problems of vagueness, overly broad applicability, and others [14]. To meet exactly the demands of the present domain and to make IPMSL taxonomic meanings precise,[1] IPMSL uses the further-specification relation, as in MERLIN [15]. The relation between an object and a further specification of it is that the latter has additional properties, or the same properties with values further specified. For instance, "5 microsec" is a further specification of "microsec", VAX-11 further specifies the concept C (computer), the VAX-11 configuration SV-AXTVA-LA further specifies VAX-11, and the package delivered to a customer includes even more specifications, eg communications equipment. This FSof can be carried through to any number of levels, eg, by further specifying over the variations that occur through the lifetime of a computer

[1]Many of the others mentioned can also be made precise, but not without a great deal more complexity. Along with this complexity of definition would come difficulty in expressing and verifying the set of allowable inferences, and in managing searches.

installation.[2] Many of the above-mentioned taxonomic systems appear inadequate for expressing such taxonomies. Of course, inheritance of properties occurs in the standard way: when some property is absent from a node, it can be inferred from the corresponding property of its ancestor (recursively).

Having the flexibility of the FSof relation allows IPMSL to take a position on two topics mentioned by Woods [28] and others. It appears to solve the "morning star" vs. "evening star" problem, attributed by Woods to Quine, and described by Woods in connection with the distinction between intension and extension. Namely, both of these phrases can be represented as separate nodes in the net, with both being further specifications of the planet Venus.[1] Many (if not all) uses of quantifiers (another problem mentioned by Woods) can be captured by simply defining nodes in the net that represent the classes defined by them. Such nodes are FSof various nodes already existing in the net. E.g., "all computers with TTL logic" might be a node that further specifies the "computer" node, with an attribute-value pair signifying its technology; such a node would have a set of FSs consisting of those satisfying the property, if one chose to store the set explicitly (one might want this to be only a temporary node, surviving only as long as a particular application demanded it). One could then add to this node any properties that hold of the class of such devices.[2] A phrase such as "some computer with TTL logic" causes the creation of a node with the appropriate FSof link, about which one could then assert further properties, and which one might eventually identify with a known machine. Both of these solution sketches depend on the ability to further specify to arbitrary levels of nestedness, and on the absence of relations that would terminate such chains (e.g. "instance"). This discussion is meant to be illustrative of the potentional strength of FS as a concept, and is thus merely suggestive - no such usages have been required in the present domain.

The distinction between objects and sets, and between set membership and set containment (subset relation), are made in many semantic net systems, but not in IPMSL. The FSof relation is best thought of as holding between typical elements of classes. Thus the distinction seems superfluous. In the rare case when there is in IPMSL's domain a reference to a collection of components, it suffices to use an ad hoc approach, eg, to attach to the "typical" element an attribute indicating a replication factor (ie, set size) - analogous things

[2]This has the danger of leading to rather deep searches; but it does seem to be strongly demanded by the domain.

[1]If these "stars" can also be Mercury, minor complications arise.

[2]There is an interaction here with the issue of how to, or whether to, distinguish a node representing a class of object from one representing a specific object. See the next paragraph.

can be done for other class concepts. Convincing arguments for the need for this distinction have not been made in the literature, though it does appear convenient for some domains. Even in those domains, however, the approach taken here seems not to sacrifice precision or inference capability.

Another taxonomic device in IPMSL is the use of compound names. Eg, "P.c VAX-11" names the "P.c" component in the "VAX-11 context". Or, "VAX-11" could be taken as further specifying (speaking informally) "P.c", eg, by referring to the particular model of P.c that it is. Note that "P.c" is itself a compound name, referring to the "c" variety of "P" (ie, central processor) - this is inherited from the original PMS language [3]. The use of compound names for methods and goals has been my practice for some time, eg, to express such ideas as verb+object ("find attribute") or noun+modifier ("tree structure"). Not only is such compounding a natural and convenient usage, but it leaves open the possibility that the system itself may somehow make use of the transparent name similarities that it gives rise to. Further, it may lead a user to be more systematic in naming. It is discussed in some detail by Martin [13] in a rather different overall approach and task. The use of such compounds is not without dangers, however, to the naive. An early design of this PMS system confused, via such compounds, the naming of components with *structural* relations - the above example of "P.c VAX-11" might lead one to make the (correct) inference that the P.c referred to is Partof VAX-11. But this immediately leads to difficulties when structures become more complex, and the structure subnet of IPMSL is more appropriate. There was also some confusion between names and a more exact *taxonomy* with respect to methods in Kernl2: often names can show local similarities, but they can also show a similar functionality but in widely varying contexts (thus appearing in different branches of the taxonomic tree). Often it is clumsy to carry out the taxonomic naming to its logical conclusion, so its use as a sole taxonomic device has been replaced by the addition of explicit taxonomic (FSof) relations. That is, the use of compound names, though convenient, should not be the only taxonomic or structural device in a system.

5.3. Contexts

The approach used here provides adequate means to a number of concepts emphasized by other researchers. Fahlman's important concept of virtual copy [6] is captured by the normal use of WM: it contains or can contain a temporary node structure that incorporates information from a number of sources. Thus a node in WM can look as if it incorporates much information which has in reality been inherited from ancestors, and processing can be done on it as if those inherited properties were immediately attached. Similar in conception is the "flat" operation in MERLIN [15, 16]. But in IPMSL this aggregating operation is deliberate and programmer-controlled, where Fahlman's system makes the "copying" automatic when information is accessed. This need for deliberate control of the inheritance may be offset by such devices as rules that take "shortcuts" in net searching, and discrimination networks that could make critical properties more directly accessible, making search more affordable for common cases. Nevertheless, this is considered to be a serious flaw in the approach, with respect to principles developed by previous AI research. It remains to be seen whether the impact on this domain will be serious. Some modifications in the basic PSA may be required.[1]

A broader meaning of the word context takes us into the area of establishing quantified scopes, partitions [11], depictions (frames) [10], and multiple worlds. Here again, it is hoped that the straightforward use of WM as a temporary, dynamic version of a portion of the network can serve the desired function, at least in the present domain. Symbols can be created in WM to bind together a number of nodes and relations, and then attributes and values associated with the symbols and thereby with the collections of nodes. WM could be[2] used for one of the possibly many worlds under consideration in some problem-solving task, say, with alternative worlds produced on demand by suitable context-updating methods or simply by evoking some other region of the permanent network and suitably modifying it to reflect local context.

5.4. Searches

Generally, control of searches in IPMSL is either by deliberate methods whose goals maintain it or by demons that respond to various unusual conditions. But a number of special topics deserve mention, since they may appear to pose peculiar search control problems.[3]

The detection of clashes and obvious inconsistencies (see [6]) requires rapid searches up the taxonomic hierarchy. Specific rules to act as shortcuts in the hierarchy are possible. That is, a rule may encode a jump in the hierarchy of several levels. If one sees this as quickly evoking information until some anomaly is detected, then it appears advantageous here that rules can express arbitrary patterns of such anomalies.

Searches with multiple origin or so-called intersection searches pose extra control problems, such as how to alternate in reasonable ways the locus of control among the various

[1]One possibility is to allow conflict resolution to fire more than one rule at a time, enabling several paths to be developed in a bounded breadth-first (beam) fashion. A more remote possibility is to build in some kind of backward chaining based on recognition-match failures, an operation common in certain applications of rule-based systems, but deliberately avoided in our approach. Considerations of practical need for the PMS domain will determine whether drastic revision in the straightforward approach will be needed.

[2]This discussion is speculative, but I think not unreasonable, given past experience with the flexibility of PSAs.

[3]This discussion of search is largely an extrapolation of present experience with IPMSL.

regions in the search. There is also a need to limit the depth of search. Again, rules provide explicit, recognition-based control. Their patterns can encompass various amounts of WM context, and can express arbitrary sorts of conditions to detect. Special searching rules with depth limits and alternations are easily specified, and fit in well with the existing net rules, essentially imposing whatever they need onto the ordinary evocation of information, dynamically. The organization of the net into the various subnets appears to be helpful in controlling things, since it allows more selectivity of information evocation.

Some systems have emphasized a need to come at the network in ways other than the usual structural and taxonomic ones. Eg, one sometimes wants to search for objects having a given property. If one desires this form of indexing in a network as implemented here, some extra mechanism is required, such as a discrimination network of properties. That such nets are readily implemented as rules has been established [18]. Otherwise, it is not difficult to envision top-down search strategies, guided by heuristics, that might be effective. Since such heuristics are expressed as rules, they could be developed interactively and tailored to particular needs (eg, avoiding life-threatening beasts, which requires rapid response).

6. Conclusions

In concluding, a few major points can be re-emphasized. Implementing the network as production rules has allowed the fruitful merging of two hitherto separate technologies: a problem-solving procedure formalism and a semantic net representation formalism. The network organizes both domain information and information about the methods of the system itself. The latter system network is usable both by an instructor of further methods (as information about existing system structures) and by the system itself in identifying items from an input sentence with the internal requirements of methods (ie, it infers which attributes to attach to values specified in an input). More advanced self-manipulation capabilities based on the network are subjects of further research. Using production rules allows the system to take advantage of recent progress in techniques for growing such systems, as is evident in the discussion of the method language commands above.

The production system architecture provides a number of useful features. Methods and network share the same working space (WM), so that rules are readily applicable both to control the evocation of network information and to provide useful information for methods to use. The network need not be entirely of the rigid formats illustrated above, but can mix the access of information with arbitrary methods. The existence of the large dynamic state makes it convenient to build elaborate temporary structures and to dynamically organize network fragments into larger units, an ability emphasized as important in many recent network designs. Rules are a natural way to express a large variety of search control strategies, an issue whose importance becomes critical in very large, diverse networks. The pattern-matching power of the rules allows such control to take into account much more than just a local search context, and affords the opportunity to integrate diverse pieces of information.

Advantages of using a semantic net implemented as production rules appear to outweigh the disadvantages, and examination of problems that can be expected when more demands are made of the system supports the continuation of this line of work. IPMSL has so far proven effective for defining, updating, and displaying a network for the DEC VAX-11 computer. Research is currently proceeding on using the basic ingredients presented here to provide IPMSL with higher-level capabilities, approaching operations that will prove useful to its intended domain of computer-aided design. It may eventually be able to improve or supplement human abilities to manipulate and maintain very complex structures, at least in certain problem areas. The work so far has advanced knowledge on at least two fronts: First, in formulating knowledge precisely so that a system such as IPMSL can encode it, one inevitably improves the basic knowledge of the domain, in organization, precision of detail, and explicitness of assumptions. Second, there is expansion of knowledge about the requirements that demanding intellectual tasks place on the production system architecture and on a larger body of AI concepts and techniques.

6.1. Addendum

As of the time of final revision of this paper, IPMSL has grown to a size of 610 rules with no degradation in its overall manageability, and with very little decrease in efficiency of interpretation. The total system size relative to the PDP-10 is becoming much more of a problem: it is expected that another 200-300 rules will consume all the remaining space addressable by a user; also the size of the present system poses a significant load for the time-sharing system, resulting in longer response times. Of the new rules, about 23% are network, as compared with 30% in the system detailed above, with the remainder being new methods. Additions to the functional capabilities include a number of new display, editing, inference, and data-checking capabilities. The system understands considerably more about the abstract attributes and values that are used to describe computers, and is able to interactively expand and correct that body of data as new computers are described. This new understanding has been applied to updating parts of the VAX-11 description that was initially entered to test the basic capabilities described above. Additions to the network part of the system include a network that holds information about attributes and values (used in data-checking), descriptions of abstract computers, and help facilities for the new methods. Construction of the higher-level methods to deal with general and specific aspects of the hardware configuration problem will begin soon. This next phase will build to a considerable extent on the existing framework.

6.2. Acknowledgments

The idea of representing a network as rules was first discussed with Allen Newell and Don Waterman. I am also indebted to Newell for suggesting and helping to formulate the task domain. General inspiration has come from the CMU Design Research Center, headed by Arthur Westerburg. Ideas on instruction are based on a long interaction with the members of the Instructable Production System group: Allen Newell, John McDermott, Lanny Forgy, Kamesh Ramakrishna, Pat Langley, John Laird, and Paul Rosenbloom.

7. References

1. Anderson, J. R., Kline, P. J. and Beasley, C. M. Jr. Complex learning processes. Yale University, Department of Psychology, July, 1978.

2. Barbacci, M. and Siewiorek, D. The CMU RT-CAD system: an innovative approach to computer aided design. *CMU Computer Science Research Review 1974-1975* (1974-1975), 39-53.

3. Bell, C. G. and Newell, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, NY, 1971.

4. Brachman, R. On the epistemological status of semantic networks. In Findler, N., Ed., *Associative Networks: The Representation and Use of Knowledge in Computers*, Academic Press, 1978. To appear. Also BBN Report No. 3807

5. Eastman, C. The representation of design problems and maintenance of their structure. IFIPS Working Conference on Application of AI and PR to CAD, North Holland, 1978.

6. Fahlman, S. *A System for Representing and Using Real-World Knowledge.* Ph.D. Th., MIT, December 1977.

7. Forgy, C. L. and McDermott, J. OPS, a domain-independent production system language. Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI, 1977, pp. 933-939.

8. Forgy, C. L. *On the Efficient Implementation of Production Systems.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, February 1979.

9. Freeman, P. and Newell, A. A model for functional reasoning in design. *Proc. Second International Joint Conference on Artificial Intelligence London* (1971), 621-640.

10. Hayes, Philip J. On semantic nets, frames and associations. Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI, 1977, pp. 99-107.

11. Hendrix, G. G. Expanding the utility of semantic networks through partitioning. Proc. Fourth International Joint Conference on Artificial Intelligence, IJCAI, 1975, pp. 115-121.

12. Knudsen, M. J. *PMSL, an Interactive Language for System-Level Description and Analysis of Computer Structures.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1973.

13. Martin, W. A. Descriptions and the specialization of concepts. MIT/LCS/TM-101, MIT, Laboratory for Computer Science, March, 1978.

14. McDermott, D. V. Artificial intelligence meets natural stupidity. *Sigart Newsletter 57* (April 1976), 4-9. ACM

15. Moore, J. A. *The Design and Evaluation of a Knowledge Net for MERLIN.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1971.

16. Moore, J. A. and Newell, A. How can MERLIN understand?. In Gregg, L., Ed., *Knowledge and Cognition,* Lawrence Erlbaum Associates, Potomac, MD, 1973, pp. 201-252.

17. Rieger, C. and Grinberg, M. The declarative representation and procedural simulation of causality in physical mechanisms. Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI, 1977, pp. 250-256.

18. Rychener, M. D. *Production Systems as a Programming Language for Artificial Intelligence Applications.* Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, 1976.

19. Rychener, M. D. Control requirements for the design of production system architectures. *SIGART Newsletter 64* (August 1977), 37-44. ACM

20. Rychener, M. D. and Newell, A. An instructable production system: Basic design issues. In Waterman, D. A. and Hayes-Roth, F., Ed., *Pattern-Directed Inference Systems,* Academic Press, New York, NY, 1978, pp. 135-153.

21. Sandewall, E. Representing natural language information in predicate calculus. In Meltzer, B. and Michie, D., Ed., *Machine Intelligence 6,* American Elsevier, New York, NY, 1971, pp. 255-277.

22. Simon, H. A. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, 1969.

23. Stallman, R. and Sussman, G. J. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence 9,* 2 (October 1977), 135-196.

24. Sussman, G. J. Slices: At the boundary between analysis and synthesis. Memo 433, MIT, Artificial Intelligence Laboratory, July, 1977.

25. Sussman, G. J. Electrical design: A problem for artificial intelligence research. Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI, 1977, pp. 894-900.

26. Waterman, D. A. Rule-directed interactive transaction agents: An approach to knowledge acquisition. R-2171-ARPA, The RAND Corporation, February, 1978.

27. Waterman, D. A. and Hayes-Roth, F. *Pattern-Directed Inference Systems.* Academic Press, New York, NY, 1978.

28. Woods, W. A. Whats's in a Link: Foundations for Semantic Networks. In Bobrow, D.G. and Collins, A., Ed., *Representation and Understanding: Studies in Cognitive Science*, Academic Press, 1975, pp. 35-82.