

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

Michael D. Rychener
Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, PA 15213

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 spatial. The importance of uniformly representing methods and network, and the importance of distinguishing temporary from permanent states are discussed. There is sketched a production system position on a number of relevant issues for advanced 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.

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 [3] and by Sussman [12]. Other AI research has discussed mechanisms that may be applicable to design systems while maintaining a general viewpoint and vocabulary, eg, Rieger's Commonsense Algorithms [9] and Moore's MERLIN [8]. The present research* aims to deal with the following problem areas:

1. Describing the basic system components.
2. Organizing those components into structures.
3. Establishing hierarchies of components and structures ranging from abstract ones to various concrete realizations.

4. Comparing structures and putting them into correspondence with each other (mapping).
5. Analyzing structures and determining effects of changes.
6. Synthesizing structures from elementary components, trying to fulfill functional specifications.
7. Coordinating multiple viewpoints of the same system.
8. Searching the design space.
9. Simulating system behavior symbolically, to ascertain dynamic properties.

An approach to these problem areas would be applicable to a wide range of systems: buildings, software systems, chemical processes, cities, etc. In addition, it could provide a central mechanism for diagnostic, tutorial, explanatory, and theory explication systems.

Computer structures at the processor-memory-switch (PMS) level are the domain of the IPMSL (Instructable PMS Language) system. The basis is that of Bell and Newell [2]. In particular, the following abstract components are considered:

*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.

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 [7] work in the PMS area is a direct predecessor to the present effort, with strong influences from Barbacci and Siewiorek [1].

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.* 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 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.

2.1. Underlying problem-solving architecture

The implementation language for IPMSL is the OPS2 [4, 10] architecture, extended 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. OPS3RX makes things look much more like "schemas" or "frames", and 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.) There are usually about 100 attribute-value sets in Working Memory (WM), each containing about five attribute-value pairs.

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 stores the facts of only one node in the network plus pointers to other nodes. When some goal 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 being executed. 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 locations.

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.

- Taxonomic. FSs and FSof relations. "FS" stands for "further specification", as used in the Merlin system [8]. An object is an FS of another if it can be viewed as the other with some additional characteristics (cf "ISA"). 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.

- Descriptive. This category includes various attributes that don't fit elsewhere, eg, cycle time, memory size, and bandwidth. Further classification may occur later.

- Physical. This includes: 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 (see [6]).

- Means-ends. Relations in this subnet directly correspond to the language for expressing methods (discussed 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 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 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. 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).

*An expanded version of this paper is available from the author, under the same title.

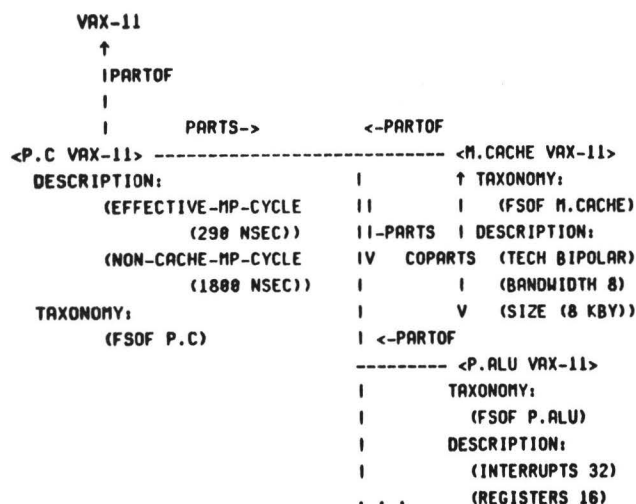


Figure 1: A fragment of a PMS network

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
KNOWLEDGE-ABOUT P.C   KNOWLEDGE-ABOUT P.C
ROLE    MAIN          NET    S0123
STRUCTURE S0136       SUBNET STRUCTURE
TAXONOMY S0138       PARTS (D.ALU M.REG)
                     PARTOF C
                     STRUCTURE P109

S0138
KNOWLEDGE-ABOUT P.C
NET    S0123
SUBNET TAXONOMY
FSOF   P
FSS    (<P.C LSI-11> <P.C VAX-11>)
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 tie together the three nodes, 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 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:

```

P100: 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 NODE 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 ITS
      STRUCTURE IS THE NEW NODE.

```

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.

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. 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 NODE
     AND THERE IS KNOWLEDGE-ABOUT THAT NODE,
     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; the 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.

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.

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.

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 (Production Memory) 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 user can evoke a region of the network and have it displayed in a tree format by using the *SHOW* command. 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 a number of options, which specify what kind of additional information is to be collected at each node.

To summarize, the basic IPMSL commands provide access (query), definition, modification, and search-and-display capabilities. Generality has been maintained: there is very little in IPMSL that is specific to the division into six types of subnets, or to the PMS domain. Also, the methods, expressed as rules, have proved to be easily modifiable. Two other design issues can be mentioned:

- The net is divided into six subnets for several reasons: to make rules a reasonable size, to avoid extra information being added to WM, to impose some structure on knowledge as it comes into the system from the user, and to modularize the methods that work with the net.

- 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 is usually significantly exceeded 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.

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 more and more 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. 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 Kernl2 approach to growing a system, ie to instruction, builds on two previous independent approaches: that of the Instructable Production System project [11] and that of Waterman's Exemplary Programming [13].* 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. Second, method construction takes place within a dynamic WM context that is similar to that in which the method will work after it is finished. Third, all of the above-mentioned method language facilities are available both to the user and to internal methods built from rules.

The basic IPMSL system is composed of 316 rules, of which about a third are network rules, another third are Kernl2 rules, and the remainder are IPMSL methods.

Some auxiliary software has been added to aid the direct coding of rules, bypassing the method language approach to growing the system whenever this seems expedient. 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.

4. ADVANTAGES AND DISADVANTAGES OF USING RULES FOR A NETWORK

My 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

production-rule systems seem to have for network systems.

- 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.

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

- 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.

- Searching methods and others can themselves be described using the same network conventions as the subject domain.

- 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 [5].

- The recognition part of the execution cycle is amenable to simple parallel implementation.

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

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. But any such network requires some space overhead.

- Search is serial: nodes are developed one at a time. But as mentioned above, given a parallel processing architecture, production systems are favorable for exploiting it.

*The interested reader should consult those references and the detailed version of this paper, available from the author.

5. 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. Using production rules allows the system to take advantage of recent progress in techniques for growing such systems.

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.

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. The work so far has advanced knowledge on at least two fronts: 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. In addition, we continue to improve our knowledge about the requirements that demanding intellectual tasks place on the production system architecture and on the whole body of AI concepts and techniques.

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. Valuable comments were made on this paper by Lanny Forgy, John McDermott, Allen Newell, and the IJCAI reviewers.

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. Of the new rules, about 23% are network, as compared with 30% in the system detailed above, with the remainder being new methods for display of information, better handling of descriptive attributes of computers, and better network access and editing capabilities.

REFERENCES

1. 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.
2. Bell, C. G. and Newell, A. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, NY, 1971.
3. 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.
4. 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.
5. Forgy, C. L. *On the Efficient Implementation of Production Systems*. Ph.D. Th., Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, February 1979.
6. Freeman, P. and Newell, A. A model for functional reasoning in design. *Proc. Second International Joint Conference on Artificial Intelligence London (1971)*, 621-640.
7. 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.
8. Moore, J. A. and Newell, A. How can MERLIN understand?. In Gregg, L., Ed., *Knowledge and Cognition*, Lawrence Erlbaum Associates, Potomac, MD, 1973.
9. 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.
10. Rychener, M. D. Control requirements for the design of production system architectures. *SIGART Newsletter*, 64 (August 1977), 37-44. ACM
11. 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.
12. Sussman, G. J. Electrical design: A problem for artificial intelligence research. *Proc. Fifth International Joint Conference on Artificial Intelligence, IJCAI, 1977*, pp. 894-900.
13. Waterman, D. A. Rule-directed interactive transaction agents: An approach to knowledge acquisition. R-2171-ARPA, The RAND Corporation, February, 1978.