The OPS Family of Production System Languages

Alexander Pasik

CUCS-232-86

Department of Computer Science
Columbia University
New York City, New York

The OPS Family of Production System Languages

Abstract

Production systems are widely used for the development of expert systems. The OPS family of languages comprise a set of low-level production system interpreters suitable for a large class of problem domains. They provide maximum flexibilty at the expense of high-level functionality such as explanation and knowledge acquisition facilities. The languages differ with respect to their expressive power and efficiency. A description of the evolution of the languages together with an analysis of their differences provides insight into the current state of the art of production system language design. While none of the languages is clearly superior to the others, it is conceivable to combine the best features of each language in a future implementation.

Table of Contents

The OPS Family of Production System Languages

1. Introduction

In the mid 1970s, many researchers in artificial intelligence were developing knowledge-based programs using production systems as a fundamental knowledge representation. Most of these scientists developed their own production system interpreters as a base for the further development of the intelligent systems. At Carnegie-Mellon University, several interpreters were being developed concurrently when, in 1976, in order to standardize the production system research and provide a stable environment for the instructable production system project [Rychener and Newell 1978], the Official Production System (OPS) was created [Forgy and McDermott 1976]. Since then, the OPS family of languages has evolved in many ways, responding to both efficiency and semantic constraints.

Today, several successful full scale expert systems, XCON at Digital Equipment Corporation [McDermott 1982], ACE at AT&T [Vesonder *et al.* 1983], and YES/MVS at International Business Machines Corporation [Griesmer *et al.* 1984], are implemented in OPS languages. Many universities use the OPS languages for expert system research. So far, the most widely used is OPS5. New production system languages are being developed based on the constructs of OPS5. Two approaches for extending OPS5's capabilities are demonstrated in YAPS at University of Maryland [Allen 1982] and Herbal at Columbia University [van Biema *et al.* 1986]. OPS5 has also been used for designing higher level expert system development environments [Pasik *et al.* 1985, Burns and Pasik 1985].

In response to the industrial acceptance of and enthusiasm for expert systems, OPS83 was developed [Forgy 1985]. OPS83 represents a departure from the earlier OPS languages in many ways, the most significant difference being that it is a true compiler. Previously, OPS production systems all ran in interpreted environments, in some instances with some degree of compilation.

The evolution of the OPS languages has resulted in languages sufficiently different from each other in their expressive power and efficiency. The history of the OPS languages and analysis of their benefits and drawbacks provides an interesting description from the points of view of both programming language design and expert systems research.

Production systems. Production systems are divided into three major sections: *production memory, working memory*, and an *inference engine* (or *interpreter*). The production memory contains a knowledge base of rules that are matched against the assertions in working memory. The inference engine performs this match, selects one or more satisfied rule instantiations to fire, and executes the actions specified. The execution of the selected rule(s) alters

2.1 OPS: Domain Independence of Production Systems

The increasingly widespread use of production systems at Carnegie-Mellon University led to the development of OPS in 1976 [Forgy and McDermott 1976]. The definition of OPS set a standard for production system languages. OPS was designed to be domain-independent. The language is a low-level production system interpreter. It provides maximum flexibility at the expense of high-level functionality such as explanation capabilities found in other systems like EMYCIN [van Melle 1979]. This philosophy was maintained through the development of all of the OPS languages giving them strength through flexibility although providing little or no semantic environment [Hayes-Roth and Waterman 1983]. OPS programs would consist of two memories: a production memory of rules and a working memory of assertions. Rules would be known as productions, and assertions as working memory elements (WMEs).

An OPS production consists of a LHS and a RHS. LHSs are a collection of condition elements (CEs) each of which is either positive or negative. There must be at least one positive CE on the LHS of each production. The CEs represent patterns to be matched against WMEs. A production is considered satisfied if and only if all positive CEs match WMEs and there is no consistent match for the negative CEs. RHSs of productions contain actions to be performed if the production is selected. These actions include those for altering working memory, input and output, and other procedural constructs.

The recognize/act cycle and conflict resolution are built into the OPS interpreter. The cycle is described as follows.

1. *Match.* Create a conflict set of all instantiations of productions with satisfied LHSs.
2. *Conflict Resolution.* Select one instantiation according to a fixed strategy.
3. *Act.* Perform the actions on the RHS of the selected production.
4. *Repeat.* Continue the cycle until no LHSs of productions are satisfied.

Working Memory Elements. OPS working memory is a set (unordered with no duplicates) of WMEs, each of which can be an arbitrarily nested list of symbols and numbers or a single symbol or number. There are no variables in WMEs.

Left-hand Sides. CEs on the LHS of production rules are abstractions of WMEs. In order to create these abstractions, pattern matching operators are provided. Constants in a pattern must match exactly the corresponding datum in the WME. The fundamental method of abstraction is the variable. The form (variable x) will match any constant in working memory. The restriction created by variables is that all occurrences of a variable on the LHS of a bindings for the b *and* c clauses consistent with (variable x) equal to 1. On the other hand, the second LHS is not satisfied because if the (variable x) is bound to 1, there is a possible match for the b clause, whereas if it is bound to 2, there is a possible match for the c clause.

Right-hand Sides. Once a production is selected for firing, its RHS is evaluated by performing each action. If an action returns a value, it is asserted into working memory. In order to remove elements, the action delete is used. Delete's argument should evaluate to a WME present in the current working memory. The RHS actions variable, segment, and quote are similar to their LHS equivalents. **Variable** returns the value bound to it on the LHS. Segment strips off the outer parentheses from its argument leaving a list of the subelements. This definition preserves the following behavior of productions: a production with identical LHS and RHS does not alter working memory if fired because the WMEs matched on the LHS are added on the RHS, but working memory contains no duplicates. The definition of **segment** above maintains this feature.

```
(-->
        (a (segment (variable x)))
        (a (segment (variable x))))
```

If working memory contains (**a b c**), the LHS of the above production would be satisfied, binding (variable x) to (b c). The RHS would evaluate to (**a b c**) because **segment** removes the outer parentheses from the binding of (variable x). Thus this production, with identical LHS and RHS, would not alter working memory. The quote action allows for the use of function names as constants. The opposite of **quote** is **eval**. Eval causes its argument to be evaluated twice. OPS provides the action **bind** for binding variables explicitly on the RHS. The action null is used to hide values returned by other actions so that they are not asserted into working memory. Input and output actions are read, **eread**, **write**, and **write&**. No provisions for file I/O exist which could indicate that the language was intended to be a prototype for future production system interpreters.

Finally, a set of actions are provided for production systems to work on their own productions. The action (readp p1) returns a copy of the production **p1**. When this return value is asserted as a WME, other productions can match on it and perform some degree of rule analysis or modification. The inverse of **readp** is **build**. It adds productions to the system. Excise takes the name of an existing production as an argument and removes that production from the system. The facility for moving copies of productions into working memory and building new productions is very flexible in OPS because WMEs are arbitrary list structures and productions are lists of three elements: the first element is -->, then the LHS, and finally the RHS.

The meanings of rules 1, 3, and 4 require further clarification. Before determining whether an instantiation has been previously fired, it is necessary to define equality of instantiations. Every time a WME is added to working memory, it is assigned a unique time tag. If the same element is added again, working memory is unchanged, but the time tag of that element is updated. An instantiation is considered identical to another if it is made of the same production matching the same WMEs with the same time tags. However, if an instantiation leaves the conflict set due to the addition of an element which contradicts a negated CE, and then returns to the conflict set due to that WME's removal, the instantiation is not considered identical, even if it is matching the same WMEs.

In rule 3, the instantiations are sorted according to the following method. The WMEs matched within each instantiation are ordered by the recency of their time tags. Then the instantiations are compared lexographically; the most recent WMEs matched by each are compared, then if there is a tie, the next most recent, until there is a winner.

Rule 4 determines specificity of instantiations by number of CEs. Rule 3 may end in a tie if the WMEs of two or more instantiations have the same time tags, and they have equal numbers of WMEs matched. Thus, for an instantiation to have more CEs but the same number of WMEs matched, it must necessarily have either more *negated* CEs or two or more CEs matching the same WME.

The final rule of conflict resolution will only be used in the rare case that two instantiations of the same production match the same set of WMEs. This will only occur if there are two distinct mappings from the same set of CEs to the same set of WMEs. In this case, an arbitrary selection is made.

As the first in the series of languages, OPS served as a test bed for many features that would appear in the later versions. The general syntax and semantics of the productions and WMEs were defined. The conflict resolution strategy and recognize/act cycles were constructed. Thus, the first of the OPS languages was built.

2.2 OPS2 and OPS4: Efficiency and Flexibility

The problem of efficiency in the match phase of production system execution led to the development of the Rete match algorithm [Forgy 1979a]. The naive approach to the match phase compares every CE from each production to each WME on every cycle. This causes the match phase to make $O(|P| \times |W|^n)$ comparisons, where $|P|$ is the number of productions, $|W|$ is the number of WMEs, and n is the maximum number of CEs on a LHS. The goal of an efficient many pattern/many object matching algorithm is to be independent of both $|P|$ and $|W|$.

Dependence on size of production memory can be reduced by compiling the productions into a network where similar CE parts are respresented once and shared between the productions. By using this in conjunction with a technique called *memory support*, dependence on the size of working memory can be reduced as well [McDermott *et al.*

user-defined predicates, the LHSs of OPS4 productions serve as very powerful pattern matching mechanisms. Nevertheless, a drawback of this flexibility is that problem-solving knowledge becomes hidden from the production system. If productions are to be dynamically modified by other productions, these meta-rules (that is, rules about rules) will not have access to the knowledge encoded in the user-defined predicates.

A related problem in meta-rule usage is caused by OPS4's format of production rules as arbitrary lists with the symbol --> within them. It is not straightforward to write a CE to match a production binding the LHS and RHS to variables. In OPS, the CE (--> =LHS =RHS) serves this purpose.

Right-hand Sides. Variables and segments on the RHS are treated as in the original OPS language. The default RHS action is addition to working memory. Also the action <add> is provided as an alternative syntax. Thus the two RHS constructs (**on box table**) and (**<add> (on box table)**) have identical function. There is also a difference in the implementation of <add>. When an existing element is added, its time tag is changed and thus its new recency can affect conflict resolution. However, this addition does not affect the first stage of conflict resolution in which instantiations already fired are discarded. The element does not count as a new element and thus previously fired instantiations involving it will not be selected. OPS4 provides an alternative RHS action <reassert>, which causes an existing element to be reasserted and thus allows instantiations involving it to be considered.

The following actions behave the same way as in the original OPS language: <delete>, <quote>, <eval>, <bind>, <null>, <write>, <write&>, <read>, <build>, and <excise>. The action <halt> is provided to halt execution of the production system. The OPS action **readp** for asserting copies of productions into working memory was not included in OPS4. This, in combination with the difficulty in matching on parts of productions, makes using meta-rules impractical.

Just as LHSs were made more flexible by the addition of user-defined predicates, RHSs in OPS4 can execute arbitrary LISP code with user-defined RHS functions. RHS functions should be defined as FEXPRs (see [MIT 1978] for a description of MACLISP) and use the OPS4 function **eval-list** to evaluate the arguments. **Eval-list** evaluates argument lists according to OPS4 syntax; it correctly interprets OPS4 variable bindings and RHS use of the segment operator.

Conflict Resolution. In OPS4, the removal of previously fired instantiations from the conflict set is termed *refraction*. It is not considered part of the conflict resolution strategy in that these instantiations are not considered part of the conflict set. The rule concerning the elimination of old WMEs is dropped from OPS4's conflict resolution. The basic strategy of recency followed by specificity is still used though the implementation is slightly different. OPS4 conflict resolution can be described as follows. The addition of rule 3 serves as an alternative measure of the specificity of an instantiation.

```
(defun <notany> (pattern data)
    (not (member pattern data)))

(defun <any> (pattern data)
    (member data pattern))


(rhs-function  <+>)
(predicate <notany>)
(predicate <any>)


(system

make-attempt-left-right (
   (side   (piece  =p1)
           (dir    left)
           (joined false)
           (color  =col)
           (curve  =cur & (<any> concave convex))
           (shape  =x))
   (side   (piece  =p2 & #p1)
           (dir    right)
           (joined false)
           (color  =col)
           (curve  #cur & (<any> concave convex))
           (shape  =y))
   -->
   (attempt (=p1 left) (=p2 right) (<+> =x =y)))
```

```
clean-unsuccessful-attempts (
    (attempt        = = (<notany> 0)) & =attempt
    (side           = = (joined false) = (curve (<notany> egde)) =)
    -->
    (<delete> =attempt))
)
```

OPS4 can be considered the first complete production system tool in the family of OPS languages. It is a flexible language with provisions for extension through user-defined predicates, variables, and functions. OPS4 runs in a LISP environment augmented with functions for the analysis, execution, and debugging of production systems.

2.3 OPS3: Sets and Attribute/Value Representation

OPS3 represents a departure from the other OPS languages. Although the recognize/act cycle and conflict resolution strategy are maintained, the similarity ends there. OPS3 uses a very different form of pattern matching as its fundamental semantics. Rather than matching by equality, it is performed by set intersection. This also implies that the fundamental data representation is the set. When matching a set within a CE (known as *varunits*) against a set in a WME (or *unit*), the match succeeds if the intersection is non-empty.

The underlying representation is sets of attribute/value pairs. The representation is more constrained in that arbitrary structures are disallowed, but is more modular and accessible in that there is no ordering imposed on the components of the sets. It is also easier to convert this representation to natural language for use in an intelligent interface. This in combination with the widespead use of this representation in other systems supported the selection of the attribute/value representation [Rychener 1980].

Working Memory Elements. OPS3 defines a more structured representation for its WMEs, referred to as units. A unit is an ***anchored set*** of attribute/value pairs. The anchor is a list of three atomic or vector values which are used for identifying information of the unit. From the point of view of the match algorithm, these values are used to index which productions are relevant when a unit is added to working memory. After the anchor, an unordered set of attribute/value pairs follows. Each attribute is an atom or vector and each value is a set. The set of pairs can be interpreted as a set of attributes each having a value; there can be no duplicate attributes even with different values. There is an implicit fourth anchor in each unit which is its time tag. It can be matched against by referring to the unit's time attribute.

Actions, on the other hand, are made of a function name (the first anchor), a variable indicating the WME being worked on (the second anchor), and arguments of the form of attribute/value pairs.

The action (!add =wme (a1 v1) (a2 v2) ...) adds the attribute/value pairs to the WME bound to the variable. If the attribute a1 exists in the =wme, this modifies the value of a1 to be the union of its current value and v1. The opposite funtion, !del, deletes attribute/value pairs from a WME. In the case of sets, the values of the attributes are modifed by taking the set difference between the current value and the value specified in the action. In order to change the values of the anchors of WMEs, the action **!mark** is supplied. The following call to !mark changes all anchors of a unit.

```
(!mark =u (pri =v1) (sec =v2) (mod =v3) (time =v4))
```

The anchors are named **pri**, **sec**, **mod**, and **time**, corresponding to the four anchors of a working memory unit. The function !rep takes a variable bound to a WME and a set of attribute/value/new-value triples. It combines the functionality of !add and !del in that it removes the values from the attributes and adds the new values to them. !Copy makes a copy of a unit. Removal of units from working memory is accomplished with the function !remove which takes an arbitrarily long list of variables bound to unit names. In order to create a new WME to further work on, the function !bind is used. The function **!keep** causes the given WMEs to be protected from deletion due to their age. !Rehearse updates the time tags of the listed WMEs. **!Time** returns the time tag of the argument WME.

On the RHS, it is often desirable to reduce the sets bound to variables to a selected member, either arbitrarily or according to certain specifications. Actions which reduce sets to atoms are **!choice**, **!minimal**, and **!maximal**. The first returns an arbitrary element, the second and third return the smallest and largest members respectively.

Actions for I/O include **!write**, **!read**, and **!dis**. The details of these actions depend on the user interface which is considered separate from the production system language.

Funtions for manipulating production memory are **!build**, and **!readpm**. Because of the difference in representations between units and productions, OPS3 provides a formal definition for a working memory unit description of a production. A production can be represented in working memory as a formally described set of units with common identifying information. When **!readpm** is executed with the name of a production as argument, a copy of the production is translated into its working memory representation and asserted. !Build creates a new rule from the units referred to in its arguments which should collectively form the description of a production.

Working Memory Representation of Productions. A sample production and its representation in working memory follow. The working memory representation is cryptic but it is necessarily so in order to adequately describe a production within the syntax of units. The unit with anchors **pm rule pm** represents an abstract form of the

Action Execution. On executing the RHS of the selected production, changes are made to working memory and the new WMEs are given their time tags is increasing order. In addition, the WMEs matched on the LHS have their time tags updated according to the following formula.

```
max(current_time_tag, present_time - 0.25 * (present_time - lowest_time))
```

This keeps relevant WMEs current. Also, whenever a WME is created, another one must be erased because of the finite size of working memory (except at the beginning of execution when working memory is not full). The oldest (by approximation) WME is the one which is deleted. OPS3 keeps track of the approximate oldest WME in the following manner. When the oldest WME is deleted, the next oldest is found by finding a WME with time tag less than the formula below.

```
previous_oldest + 0.25 * (present_time - previous_oldest)
```

This technique reduces the search time to find the actual oldest WME. An exception to this method of finding the next element to be deleted is that every WME has a reference count (that is, a count of how many other WMEs refer to it). A unit with a non-zero reference count will not be selected for deletion unless all units are referenced.

Sample Production System. In the OPS3 implementation of the jigsaw puzzle, its set matching mechanism eliminates the need for the user-defined predicates `any` and `notany`. Nevertheless, due to its lack of provision for user-defined predicates, the calculation for fit must still take place on the RHS.

```
make-attempt-left-right [
    (side =p1 left =                 (joined false)
                                     (color  =col)
                                     (curve  (=cur convex concave))
                                     (shape  =x))
    (side (=p2 #p1) right =          (joined false)
                                     (color  =col)
                                     (curve  (#cur convex concave))
                                     (shape  =y))
    -->
    (attempt =p1*left =p2*right      (fit (!plus =x =y)))      ]
```

which have been essentially ignored in the further evolution of the OPS family. One of its contributions, however, that of attribute/value pairs, was adopted in OPS5 which has become the most widely used of the OPS family of languages.

2.4 OPS5: Sacrificing Flexibility for Speed

OPS5 is generally considered one of the most widely used production system languages in both universities and industry. After the initial OPS4 implementation, Digital Equipment Corporation's R1 was translated and further developed in OPS5. OPS5 was considered by the developers to be easier to use and more intelligible than its precursor. There was, however, another reason underlying the reimplementation. The intuitive advantage of using production systems is the incremental nature of the development of such programs. Nevertheless, it was found that in practice writing production systems in OPS4 and OPS5 did not reflect this advantage. Production rules are often not autonomous, leading to convoluted, difficult to maintain programs. Thus reimplementation also served to restructure the program on the basis of having a clearer picture of the system, regardless of the change in language [McDermott 1981].

OPS5 was chosen, however, by the R1 developers as well as by many others for expert system development [Griesmer *et al.* 1984, Stansfield 1986]. This acceptance indicates that there is some rationale for the choice. OPS5 provides a greatly needed speed improvement over OPS4 and the adoption of the attribute/value representation is encouraging from the standpoint of the structure's widespread use in the artificial intelligence community. OPS5 was originally implemented in MACLISP [MIT 1978], Franz LISP [Foderaro 1980], and BLISS [Digital Equipment Corporation 1980] making it available to a wide audience. Its debugging facilities are more robust than those of previous OPS languages making it more suitable for the development of larger systems [Forgy 1981].

Working Memory Elements. The underlying structure of a WME in OPS5 is a one-dimensional array of atoms (numbers or symbols). There can be no nested lists or scalar atoms in working memory. However, OPS5's syntax provides a mechanism for the use of attribute/value pairs within this framework. Prior to compiling productions, the attributes of different classes of WMEs are declared using the literalize statement. After processing all literalize declarations, OPS5 assigns a unique integer index to each attribute mentioned.

```
(literalize goal        status type object)
(literalize object      type color size)
(literalize location    x y)
```

Right-hand Sides. OPS5 provides a more rigid syntax for its RHSs as well as its LHSs. A RHS pattern is defined as a sequence of values or attribute/value pairs where the values are either constants, variables, or function calls. There are exactly 12 RHS actions provided many of which take RHS patterns as arguments.

In order to add elements to working memory, the action **make** is provided. **Make** takes a RHS pattern as its argument. The OPS5 interpreter uses a special array of 127 elements called the result element during RHS evaluations. It is not in working memory, rather it is used for intermediate processing in RHS actions. First, the result element index is initialized to 1. Then the RHS pattern is scanned; each value is placed into the result element at the current index and each attribute causes a reassignment of the current index. During RHS pattern evaluation, constants evaluate to themselves, variables are replaced by their bindings, and RHS functions (as opposed to actions) can be called. OPS5 provides a set of functions and the facility for users to integrate their own into the system. After the RHS pattern is evaluated into the result element, **make** asserts it into working memory.

The **remove** action removes elements from working memory. Its arguments are either element variables or numeric values indicating which positive CE on the LHS matched the WME to be removed. OPS5's action **modify** is equivalent to a **remove** followed by a **make**. Its arguments are first an element designator (number or variable) and then a RHS pattern. **Modify** first removes the element indicated, then fills the result element with a copy of it. Then a RHS pattern evaluation occurs, overwriting slots in the result element as indicated. Finally, the result element is asserted into working memory. The actions **bind** and **cbind** are provided for explicit binding of variables on the RHS. Actions for I/O are **write**, **openfile**, **closefile**, and **default**. The action **default** allows the user to specify which is the default file for I/O. This is initally set to be the user's terminal. The action **halt** halts the system.

OPS5 provides the **build** action for building new productions. Nevertheless, because of the different structures used to represent WMEs and productions, and the lack of any formal definition for translating information between them, it is impractical to use meta-rules in OPS5.

The **call** action is used to invoke user-defined actions (as opposed to user-defined functions which are called within other actions). A call to this action has the form (**call** ***name pattern***) where the name is that of the user-defined action and the pattern is a RHS pattern. The pattern is evaluated into the result element and then the function **name** is called.

LISP interface. OPS5 provides mechanisms for users to define their own actions and functions. Actions are invoked using **the call action**, and functions can be called within RHS patterns. It is important to note that all interaction between the user-defined routines and OPS5 must take place via the result element. Actions must be defined as EXPRs of no arguments. Upon invocation of a user-defined action, the result element will already contain the evaluated pattern argument of **call**. In order to access these values within the routine, OPS5 provides the LISP function **$parameter**.

the cause of implementing the attribute/value representation on top of the fast underlying array structure; the attribute/value representation is a syntactic convenience for the user, not the actual representation.

Sample Production System. The jigsaw puzzle system in OPS5 is similar to the OPS3 version. Both use the attribute/value representation but OPS5 uses its disjunction and conjunction operators where OPS3 relied on its set operations.

```
(literalize side       piece   dir     joined
                       color   curve   shape)

(literalize attempt    piece1  piece2
                       dir1    dir2
                       fit)

(p  make-attempt-left-right
    (side    ^piece   <p1>    ^dir    left
             ^joined  false
             ^color   <col>
             ^curve   { <cur> << convex concave >> }
             ^shape   <x>)
    (side    ^piece   { <p2> <> <p1> }      ^dir     right
             ^joined  false
             ^color   <col>
             ^curve   { <> <cur> << convex concave >> }
             ^shape   <y>)
    -->
    (make attempt  ^piece1  <p1>    ^piece2  <p2>
                   ^dir1    left    ^dir2    right
                   ^fit     (compute <x> - <y>))  )
```

```
(p  clean-unsuccessful-attempts
    (attempt        ^fit      <> 0)
    (side           ^joined false
                    ^curve <> egde)
    -->
    (remove 1))
```

OPS5 is a relatively high speed production system interpreter. It achieves this speed by combining the fast pattern matching techniques of the Rete algorithm with fast data structures at the expense of flexibility. LHSs are restrictive by disallowing any user-defined constructs. The LHSs therefore are compiled into an efficient network. The RHSs, however, are still interpreted.

2.5 YAPS: Structured Flexibility and Multiple Production Systems

In response to the diminished flexibility of OPS5, YAPS was developed in order to provide a production system interpreter with the speed of OPS5 yet with substantial increases in expressive power [Allen 1982]. The major objections to OPS5 addressed by YAPS are the flat structure of WMEs, the lack of user-defined LHS predicates, the interpretation (as opposed to compilation) of the RHS, and the difficulty in running a production system under another program's control. YAPS also provides a mechanism for implementing a set of discrete production systems with a method for communication between them.

Working Memory Elements. YAPS abandons the use of the attribute/value representation and reverts to the arbitrary list structures used in OPS4. However, atomic WMEs are not allowed. In order to handle this representation, however, it is necessary to re-introduce the segment operator for matching lists of indefinite length. Working memory as a whole is more similar to OPS5 in that duplicate elements are supported with their time tags being the unique identifying information.

Left-hand Sides. YAPS separates LHSs into two parts: the first part has patterns to be matched against WMEs, the second part performs tests on the matched data. Thus the use of predicates, not variables, pattern-and, or any other pattern-matching functions is isolated from the actual match process. The match part only contains descriptions of WMEs with constants, variables, and segmentation specified. Variables are distinguished by a leading -. The segment

Multiple Production Systems. YAPS supports the use of multiple production systems. Several independent systems of production and working memories can co-exist in the environment [Allen 1983]. They can communicate using a standard mechanism developed at the University of Maryland which supports object-oriented programming [Allen *et al.* 1983]. The construct (**<- *name function-description***) indicates that the function described should be sent to the named production system for evaluation. Thus, if the firing of a production should cause the WME (a b c) to be asserted in the working memory of another system the name of which is bound to **-name**, its RHS should contain the expression (<- **-name** '**fact** '(a b c)). Most of the functions provided by YAPS have equivalent forms using this message passing construct. The top-level functions can also be directed to a named system using the same facility. YAPS has top-level functions for displaying productions, WMEs, or the conflict set: **printp**, **db**, and cs respectively. Also, OPS5 style tracing is supported with the functions **trace**, and **pbreak**.

Conflict Resolution. YAPS's conflict resolution strategy is similar to OPS5's MEA. Rather than preferring instantiations with more recent matches to primary CEs, it prefers instantiations matching more recent **goal** elements. Other than this, the conflict resolution strategy is the same.

Sample Production System. The jigsaw puzzle system implemented in YAPS is different from the other versions because of its powerful testing mechanism on the LHS. By writing the appropriate LISP functions, the system is reduced to a single rule.

```
(defun opposite (side1 side2)
    (or     (and (eq side1 'left) (eq side2 'right))
            (and (eq side1 'top) (eq side2 'bottom))))


(defun inverse (x y)
    (zerop (+ x y)))
```

assimilation into other programming paradigms achieved by the message passing features. They allow production systems to be used in conjunction with object-oriented programs without necessarily imposing a dominance relationship between the methods. Either type of program can control the other through the general message passing facility provided.

2.6 YES/OPS5: Making OPS5 Suitable for Real Time Applications and Monitoring

At IBM's Thomas J. Watson Research Center, the YES expert system project selected OPS5 for the development of YES/MVS: a production system designed to monitor an IBM mainframe running the MVS operating system. Although most of the needs of the project were satisfied by OPS5's features, some functions were added to make it more practical for use in a real time monitoring enivironment [Griesmer *et al.* 1984].

New Actions. An important part of operating a large computer system is scheduling. This task often requires that certain functions be performed at a later time. For this purpose, the action **timed-make** was created. Calls to this action take two forms, specified below. The former indicates that the described WME whould be asserted at the absolute time specified. The latter gives a relative time at which the WME should be added.

(timed-make ***RHS-pattern*** (AT ***time-description***))
(timed-make ***RHS-pattern*** (IN ***time-description***))

YES/MVS was built as separate component production systems. A method for their communication was provided with the action **remote-make**. This action has the same syntax of **make** but an additional attribute **^Rm-to:** is specified with the name of the destination production system supplied as its value.

Finally, the action **ops-wait** suspends the recognize/act cycle until a **timed-make** or remote-make affects the working memory.

Recognize/Act Cycle Change. In order to implement the above new actions, an extra step is placed in the recognize/act cycle. After the actions of the selected production are processed, the interpreter checks if there are any timed-make or **remote-make** actions to be processed in this cycle. If so, the new elements are asserted before the next cycle.

Working Memory Elements. OPS83 WMEs are record structures in which the fields are either scalars, arrays, or records, but not other elements. A given element type is declared in a type declaration statement as in the example below.

```
type    goal = element (
                 status  : symbol;
                 type    : symbol;
                 value   : real;
                 loc     : array(2:integer);
        );
```

Left-hand Sides. LHSs of OPS83 rules are compiled into a Rete network. Thus they are not executable pieces of code, rather they are declarative descriptions of possible states of the working memory. They are referred to as *simple contexts*. A simple context is a portion of an OPS83 program which cannot refer to or change global variables, perform any I/O, or alter working memory. Data comparison, calls to procedures or functions that are also simple contexts, and pattern matching against WMEs comprise the LHSs of rules.

The LHS is made of a sequence of CEs at least one of which is positive. Negative CEs are preceded by a ~ with no provision for conjunctions of negations. Any positive CE can be preceded by a variable name, in which case that variable is bound to the matching WME upon LHS satisfaction.

CEs have the following form. They are enclosed in parentheses with a symbol indicating the element type at the head. Then there follows a sequence of terms, each of which is either an expression in parentheses, a function call of type logical, or a test on a field of the element. OPS83 provides the *pseudo-variable* @ to refer to the element being matched. Following the CEs is an optional *rating expression* enclosed in square brackets. The expression can refer to the variables bound on the LHS and must evaluate to a real number. The value for a given instantiation can be accessed during the user's conflict resolution strategy or recognize/act cycle. For example, the LHS containing the CE above could be followed by the expression [**&G.value / 100.0**].

Make, Modify, Remove, and On Statements. These working memory altering statements can appear in any non-simple context. They most often occur in portions of a program that initialize working memory and on RHSs of rules. The make statement asserts a WME into working memory concurrently altering the conflict set to reflect the addition. Remove removes a list of WMEs an alters the conflict set accordingly. **Modify** removes the specified element and asserts a copy with the specified fields changed.

```
function select () : integer
{
        local   &n      : integer,      &ch     : integer,
                &sp     : integer,      &time   : integer,
                &w      : integer,      &u      : integer,
                &r      : real,         &i      : integer,
                &wc     : integer,      &cc     : integer;

        &n = cssize(); &ch = -1; &sp = -1; &time = -1;
        for &i = (&n downto 1)
                if (    instance(&i,&w,&u,&r,&wc,&cc)  Λ
                        &u = 0 Λ
                        ((&w > &time) V
                         ((&w = &time) Λ (&cc > &sp))))
                {
                        &ch = &i;
                        &sp = &cc;
                        &time = &w;
                };
        return(&ch);
};

procedure run ();
{
        local   &i      : integer;
        &i = 1;
        while (&i > 0)
        {
                &i = select();
                if (&i > 0) fire &i;
                if (&haltflag) return;
        }
};
```

```
rule fitpieces
{
&side1 (side   joined = 0B;
               (@.curve = convex V @.curve = concave));
&side2 (side   joined = 0B;
               piece <> &side1.piece;
               opposite(&side1.dir,@.dir);
               color = &side1.color;
               (@.curve <> &side1.curve);
               (@.curve = convex V @.curve = concave);
               inverse(&side1.shape,@.shape));
-->
        write () |Joining |,    &side1.dir, | side of |, &side1.piece, | with |,
                                &side2.dir, | side of |, &side2.piece, '\n';
        modify &side1 (joined = 1B);
        modify &side2 (joined = 1B);
};

};
```

OPS83 is a language directed at industries interested in the commercial development of expert systems. It creates fast systems because it compiles the code fully. Both procedural and rule-based programming paradigms are supported giving a high degree of flexibility to the language. However, the language is not suitable for artificial intelligence research. It lack the flexible environment associated with interpreted systems. It has no provisions for introspection and learning. Nevertheless, OPS83 is an adequate programming tool for producing fast, static expert systems for industrial applications.

		OPS	OPS2	OPS4	OPS3	OPS5	YAPS	YES	OPS83
WM Structure	Arbitrary Structures	■	■	■			■		■
	Attribute/Value Scalar					■		■	
	Attribute/Value Set				■				
Match	By Equality	■	■	■		■	■	■	■
	By Set Intersection				■				
Flexibility of Code	LHS User Predicates			■			■		■
	RHS User Actions		■	■	■	■	■	■	■
	Conjunction of Negs	■	■	■	■		■		
	User Conflict Res								■
Learning	Build Productions	■	■	■	■	■	■	■	
	Read Productions	■			■				
Others	Multiple Systems						■		
	Timed Makes							■	
	Commun/Interrupt			■			■	■	

■ Language fully implements this feature.

■ Language has a restricted form of this feature.

OPS3 retires WMEs when they are old. OPS4 provides a facility for this feature (modified from the earlier versions which automatically deleted old elements). Although this assures the use of working memory as short term, other production system programming techniques have been studied which make use of portions of working memory as another source of problem-domain knowledge which should thus not be deleted [Pasik and Schor 1984]. These and other uses of long term WMEs have been analyzed [Brownston *et al.* 1985].

```
(p first-rule
        (block ^size <s>)
        -->
        (make temp ^value (f <s>)))

(p second-rule
        (temp ^value < 100)
        -->
        (write Found a block with f(size) less than 100)
        (remove 1))
```

Conjunctions of negated CEs is available in all the languages except OPS5 and OPS83. This is a more serious drawback in the languages because in order to circumvent the restriction it is often necessary to change the data structures used. The only way to negate a conjunction of conditions is for those conditions to be part of a single WME. Another alternative is to separate the problem into two rules. For example, a production is necessary for the implementation of round robin scheduling. In this, the following condition is required of the next user to be scheduled.

$$\exists_{user} \mid (\text{number-scheduled(user)} = m) \wedge$$
$$\sim \{ \exists_{user} \mid [(\text{number-scheduled(user)} < m) \wedge (\text{job-waiting(user)})] \}$$

This means that for a user to be selected for scheduling with m jobs already scheduled, there must be no other user with fewer than m jobs scheduled with a job waiting. In OPS3, which allows conjunctions of negations, this could be expressed as follows.

```
P1 [    (job entry =j =c1          (status unscheduled)
                                   (user =u))
        (user id =u =c2            (number-scheduled =m))
        &-      (user id =u2 #c2          (number-scheduled (!neg =m)))
                (job entry =j2 #c1        (user #u)
                                          (status unscheduled))          -&
        -->
        (!rep =c1 (status unscheduled scheduled))
        (!rep =c2 (number-scheduled =m (!plus =m 1)))          ]
```

jobs pending. An alternative approach involving changing the structures used to represent users. In addition to keeping how many jobs have been scheduled for each user, a count of how many unscheduled jobs is kept as well. Since both these values are kept within the same WME, a conjunction of conditions on these values can be negated.

```
(p P1
        (job-entry        ^id <j>
                          ^status unscheduled)
        (user-id          ^id <u>
                          ^number-scheduled <m>
                          ^number-unscheduled <n>)
        -      (user-id          ^id <> <u>
                                 ^number-scheduled < <m>
                                 ^number-unscheduled > 0)
        -->
        (modify 1         ^status scheduled)
        (modify 2         ^number-scheduled (compute <m> + 1)
                          ^number-unscheduled (compute <n> - 1)))
```

Another aspect of code flexibility is provided in OPS83. In this language, the user can develop a recognize/act cycle and conflict resolution tailored to the specific needs of the system. The problem of procedural control of production systems is addressed with this flexibility. This control can take the form of a recognize/act cycle which performs a step prior to conflict resolution. This step would limit the eligible instantiations to those derived from a production in an *active set*, as determined by an analysis of the current problem state [Georgeff 1982]. The inability to adequately separate control knowledge from other domain knowledge has been a hinderance to OPS5 users [Ennis 1982]. By providing this flexibility, OPS83 has the potential to be used for a wide range of domains.

3.4 Learning Capability

There are both syntactic and semantic requirements of production system languages for learning to be practical. By learning, it is meant that the production system should be able to automatically create new productions through an analysis of its existing knowledge structures. The only syntactic construct necessary is the build action; actions to in the jigsaw puzzle example, the expressive power of YAPS and OPS83 surpasses that of their predecessors in their ability to perform complex tests during pattern matching. OPS4's support of user-defined LHS predicates is limited in that variables cannot be passed as arguments. OPS3 and OPS5 have no provision for the custom pattern matching needed for the problem.

The attribute/value formalism used in OPS3, OPS5, and OPS83 makes productions more concise, readable, and modular. The combination of this representation and the set matching in OPS3 is a very appealing mechanism. However, OPS3 falls short by not providing adequate features for extending the language.

OPS83 provides great flexibility. Its support of LHS predicates and arbitrary RHS bodies is complemented by its flexible mechanism for the implementation of conflict resolution strategies and recognize/act cycles. Nevertheless, OPS83's practicality is diminished because of its lack of environment. Debugging OPS83 is extremely difficult because custom trace routines must be written for each program. Also, the programs must be recompiled on each development iteration.

On analyzing these languages, the importance of certain features in production systems becomes evident. None of the languages is clearly superior to the others. A powerful production system language can be envisioned providing the following features.

- the attribute/value formalism of OPS3, OPS5 and OPS83
- the set matching capability of OPS3
- the powerful data testing facility of YAPS and OPS83
- the rich, interpreted environment of OPS4 and OPS5
- a compiler for greater speed of completed systems as in OPS83
- the customizable conflict resolution and recognize/act cycle of OPS83
- the self-representational capability of OPS3
- the communication primitives for distributed systems of YAPS and YES/OPS5

Clearly, such a language does not exist. Nevertheless, it is not an inconceivable combination. The next generation of production system languages will likely use many of these features in combination with constructs for the natural representation of parallel match functions and actions.

References

Allen E. (1983) *YAPS: A Production Rule System Meets Objects.* AAAI-83, 5-7.

Allen E. (1982) YAPS: Yet Another Production System. Technical Report, Department of Computer Science, University of Maryland.

Allen E., Trigg R., and Wood R. (1983) Maryland Artificial Intelligence Group Franz LISP Environment. Technical Report, Department of Computer Science, University of Maryland.

Brownston L., Farrell R., Kant E., and Martin N. (1985) *Programming Expert Systems in OPS5.* Reading Massachusetts: Addison Wesley.

Burns L.M. and Pasik A. (1985) A Generic Framework for Expert Data Analysis Systems. Technical Report, Department of Computer Science, Columbia University.

Digital Equipment Corporation. (1980) BLISS Language Guide.

Ennis S.P. (1982) *Expert Systems: A User's Perspective of Some Current Tools.* AAAI-82, 319-321.

Foderaro J.K. (1980) The Franz LISP Manual. University of California at Berkeley.

Forgy C.L. (1979a) *On the Efficient Implementation of Production Systems.* Ph.D. Thesis, Carnegie-Mellon University.

Forgy C.L. (1979b) OPS4 User's Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

Forgy C.L. (1981) OPS5 User's Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

Forgy C.L. (1985) OPS83 User's Manual and Report. Production Systems Technologies.

Forgy C.L. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19(1)*: 17-37.

Forgy C.L. and McDermott J. (1977) *OPS, a Domain-independent Production System Language.* IJCAI-77, 933-939.

Forgy C.L. and McDermott J. (1976) OPS Reference Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

Forgy C.L. and McDermott J. (1978) OPS2 Reference Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

Georgeff M.P. (1982) Procedural Control in Production Systems. *Artificial Intelligence 18(2)*: 175-201.

Griesmer J.H., Hong S.J., Karnaugh M., Kastner J.K., Schor M.I., Ennis R.L., Klein D.A., Milliken K.R., and VanWoerkom H.M. (1984) *YES/MVS: A Continuous Real Time Expert System.* AAAI-84, 130-136.