Towards the Formal Specification of an OPS5 Production System Architecture

David I. Gold
P. A. Consulting Group, Buckingham Palace Road, London SW1W 9SR, England

R. T. Plant
Dept. of Computer Information Systems, University of Miami, Coral Gables, Florida 33124

The article presents a formal specification for many important aspects of the OPS5 production systems framework. The article illustrates how an abstract formal specification of a production system can be created and the benefits this provides to those involved in the development of knowledge-based systems. The formal specification is preceded by an informal specification of a production system upon which the formal model is based and the development is illustrated through the use of concrete examples. The notation used is that of "Z" (J. M. Spivey, *The Z Notation*, Prentice-Hall, Englewood Cliffs, NJ, 1990), a language based upon typed set theory. This language has been used to success in the specification of critical conventional software systems (I. Hayes, Technical Monograph PRG-46, Oxford University Computing Laboratory, Oxford, England, 1985) and which is formal enough to allow for the creation of rigorous specifications, yet is of a form that makes these specifications "readable." The aim of the article is to show that formal techniques can be applied to areas of knowledge-based system development, thus promoting correctness, reliability, and understanding. © 1994 John Wiley & Sons, Inc.

I. FORMALITY IN SOFTWARE DEVELOPMENT

The need for correct computer systems has long been recognized and researchers have become focused upon the need for formality in all stages of system development. In no stage is this more important than in that of specification, where the basis for all subsequent development takes shape. It is vital that the specification be clear, concise, and above all unambiguous in its presentation of the system requirements. The specification acts as an intermediary for both systems analyst and the person for whom the system is being constructed. Therefore, the need for a specification that is capable of being read by both parties is needed, a goal that is not always an easy one to satisfy, given that the specification needs to be formal enough to avoid the ambiguities ren-

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 9, 739–768 (1994)

CCC 0884-8173/94/080739-30

dered by the use of a natural language specification, and given the constraint that most people instigating the creation of a system do not have a training in formal techniques. The second role that a specification fulfills is as a baseline document that the programmer will develop his system from, again this requires that the programmer be literate to formal techniques. After taking these personnel constraints into account, the advantages that can be gained through the use of rigorous and formal methods can be enormous in all stages of development. Firstly, the act of creating a specification forces all parties to agree upon a document that unambiguously defines the software that to be produced, and that the software match the specification. This avoids the most costly of all errors, programming errors due to the misinterpretation of a specification, as with a formal specification there is no room for misinterpretation. Secondly, formal techniques accommodate the development of a concrete specification from the abstract one through the use of data refinement rules,[1] leaving the programming of the system as a trivial exercise. Thirdly, the specification can act as a mechanism to assist in the maintenance of the system.

It can therefore be seen that the creation of a formal specification can assist a system to reach a degree of correctness unobtainable for systems created without employing such mechanisms.

II. FORMALISM FOR KBS

The use of formal techniques in computer science has its detractors who state that the overheads imposed in the creation of formal specifications are not acceptable, for systems that are not trivial. This argument can be countered in several ways, by showing examples of large system specifications,[2] giving a cost benefit analysis of the costs saved by not having to rewrite a large system due to problems found in a nonformal specification or by considering the need for correctness in critical systems such as those used to control nuclear reactors or avionic systems. In systems where correctness is an issue, the use of formal techniques had, due to these factors, become more prevalent in recent years. It follows therefore that the area of knowledge-based systems, in which sensitive and critical software is often constructed, is an area that will benefit from the deployment of such techniques.

Knowledge-based systems can be created in many different ways, for example the representation may be based upon the use of frames, semantic networks or rules. The control architecture may be forward chaining, backward chaining or bi-directional and the knowledge itself may necessitate the use of a heuristic method such as certainty factor algebra, fuzzy logic or probability measures. Each of these areas and their interaction with each other can benefit from formalization, as there has been little work to provide even a rudimentary denotational semantics for the major representations. It is therefore the aim of this article to show how an abstract specification of a production system can be created and how this will benefit knowledge engineers and expert system builders. We will first present an informal specification of a production system, this will be followed by a brief introduction to the approach we take towards formalization before presentation of the specification itself.

III. AN INFORMAL SPECIFICATION OF A PRODUCTION SYSTEM

We now present an informal description of a *production system*, prior to the development of an abstract specification which describes this class of systems.

A production system has three components: (i) A *working memory* which represents the current state of the problem. (ii) A collection of *productions* which are stored in a production memory. A production consists of a *condition* part and an *action* part. The condition is some pattern which is either *satisfied* or not satisfied by the current contents of the working memory; the action part specifies some change to be made to the working memory when the condition part is satisfied. (iii) An interpreter which executes the production system using a *recognize-act* cycle which may be described by the following algorithm:

```
While (There is at least one production
                    whose condition part is satisfied by the
                    contents of the working memory) and
                    (the goal* has not been reached)
do
                    select a satisfied production and carry
                    out its action part
od
```

The set of productions whose condition part is satisfied on any one particular cycle is known as the *conflict set*. Note that this algorithm is nondeterministic, since we have not specified how we should choose a production from the conflict set when more than one is available. This process of selecting which production to execute, if more than one is eligible, is known as *conflict resolution*.

IV. A SIMPLE EXAMPLE OF A PRODUCTION SYSTEM

We can illustrate these ideas by means of a simple example, the Water Jug Problem.[3] Suppose we are given two jugs, one of which holds four gallons of water and one which holds three gallons. The problem is to fill the four-gallon jug with exactly two gallons of water, given that we are only allowed the following operations: fill a jug, empty a jug or transfer (some/all of) the contents of one jug to the other.

We can solve this problem by means of a production system. As we stated earlier, the working memory should represent the current state of the problem—in this case, the number of gallons in each jug. We can represent this by the ordered pair (x, y) in which x is the number of gallons in the four-gallon jug, and y is the number in the three-gallon jug; the *goal* may then be represented by the pair $(2, y)$. Some typical productions might be:

*Typically, there will be a *goal—a solution* to a particular problem for which the production system has been written. This will be represented by a certain configuration of the working memory.

p1 $(x, y \mid y < 3) \rightarrow (x, 3)$	Fill the 3-gallon jug
p2 $(x, y \mid x > 0) \rightarrow (0, y)$	Empty the 4-gallon jug
p3 $(x, y \mid x + y >= 4$ and $y > 0) \rightarrow (4, y - (4 - x)$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
p4 $(x, y \mid x + y <= 4$ and $y > 0) \rightarrow (x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug

Here, the *conditions* (to the left of the arrows) are terms related to the current values of x and y. The *actions* (to the right of the arrows) specify a new working memory which has been transformed in some way corresponding to one of our primitive operations. Obviously, we want to apply some sequence of productions which result in x having the value 2 (we are not concerned with the final value of y). In fact, the productions which we have listed above are sufficient to find a solution—the sequence p1;p4;;p1;p3;p2;p4 will leave exactly 2 gallons in the 4-gallon jug (assuming that both jugs are initially empty).

Note that in any given state, there may be many applicable productions that it is quite possible for the system to enter trivial infinite loops, e.g., repeatedly filling and emptying one of the jugs. The conflict resolution strategy should handle such potential problems and enable the interpreter to choose sensibly from the conflict set so that the system makes progress towards the goal.

V. THE FORMAL SPECIFICATION OF A PRODUCTION SYSTEM

In this section, we shall develop an abstract formal specification of production systems which is based on the ideas which we have described. The "Z" notation[4,5] in which this specification is presented, is based upon typed set theory and has been used to construct specifications for real world critical systems based upon conventional programming techniques.[6]

In this specification we shall not be concerned with the internal structure of working memories or productions; we therefore choose to introduce two given sets, WM—the set of all possible working memories, and PRODUCTION—the set of all possible production rules.

We can model a production rule by a relation (it might seem, the light of our simple example system, that we could model a production as a *function,* but later in the article we show that in some systems, the productions require a relational model) which maps working memories onto (appropriately transformed) working memories, the domain of each such relation being exactly those working memories which satisfy the condition part of the production. By appropriately transformed, we mean that each new working memory must be obtainable from the old one, as a result of carrying out the action part. We can define a function which produces the relation that a particular production rule "represents":

$$\mathtt{rel} : \mathtt{PRODUCTION} \rightarrow (\mathtt{WM} \leftrightarrow \mathtt{WM})$$

e.g., with reference to our simple example system:

$$(4,0)\mapsto(4,3) \in rel\ (p1) \quad \& \quad (3,3)\mapsto(4,2) \in rel\ (p3)$$

A production memory is some set of productions:

$$pmemory : (\mathbb{P}\ PRODUCTION)$$

The system state is composed of a working memory, and a production memory:

STATE

$$wmemory : WM$$
$$pmemory : (\mathbb{P}\ PRODUCTION)$$

We define a ΔSTATE to be a state–state transformation in which the production memory remains unchanged:

ΔSTATE

$$STATE$$
$$STATE'$$

$$pmemory' = pmemory$$

In the subsequent specifications, we shall always assume that a ΔSTATE is of this form. The recognize-act cycle can be modeled by a ΔSTATE in which the old working memory does not satisfy the *goal* and the new working memory is obtained by applying some production, whose condition is satisfied in the old state, to the old working memory:

RACYCLE

$$\Delta STATE$$

$$\neg\ G\ (wmemory)$$
$$\exists\ p : pmemory \bullet (wmemory, wmemory') \in rel\ (p)$$
WHERE G is some predicate which specifies the goal

Note that we do not specify which productions would be executed if more than one is eligible, i.e., we do not specify a conflict resolution strategy, and therefore, our specification is nondeterministic.

VI. THE FORMAL SPECIFICATION OF A PRODUCTION SYSTEM FRAMEWORK

In this section we shall develop a specification of a production system framework, in which production systems for various applications may be built. The system will be based on some of the ideas used in the OPS5 production system.[7,8] This has particular relevance to the area of expert systems, since R1, one of the best known and most successful systems is written in OPS5.[9] The specification will illustrate how the ideas introduced in the previous section might be implemented in a particular system.

A. The Working Memory

The elements appearing in the working memory are record-like structures called *attribute-value elements*, which consist of a *class-name*, followed by some *attributes* and their respective *values*, suppose we decide to try and solve the water jug problem using this system. An empty 4-gallon jug must be represented:

CLASSNAME	ATTRIBUTE NAMES	ATTRIBUTE VALUES
jug	capacities	4
	contents	0

Here the class-name is jug, the attribute-names are capacity and contents, their respective values being 4 and 0.

In a world of colored building blocks, we might want to represent a green block whose sides are all of length 10:

block	color	green
	length	10
	breadth	10
	height	10

or a red block with sides of various lengths:

block	color	red
	length	4
	breadth	5
	height	6

We need to be able to differentiate between class-names, attribute-names, and values in our specification, and therefore introduce three given sets: CLASS—the set of all class-names, ATTR—the set of all attribute-names, and VALUE—the set of all values.

All attribute-value elements must be *declared* before they can be used—a declaration consists of a class-name, and the set of attributes which are associated with this class-name, e.g.,

block	color
	length
	breadth
	height

We can model the set of declared attribute-value elements by a function which maps a class-name onto their associated collection of attributes:

```
declared : CLASS ⇸ (ℙ ATTR)
```

so, that, e.g., declared (block) = {color, length, breadth, height}

We can now describe an attribute-value element more formally:

```
AV-ELT
class : CLASS
avpairs : ATTR ⇸ VALUE
---
class ∈ dom declared
dom avpairs = declared (class)
```

In this system, a working memory is some collection of these attribute-value elements, so that WM—the set of all possible working memories, is the set of all possible sets of attribute-value elements, i.e., ℙ AV-ELT

```
WM
ℙ AV-ELT
```

The following is an obvious lemma:

$$\forall\, m : \mathrm{WM}$$
$$\forall\, x, y : m.$$
$$x.\mathrm{class} = y.\mathrm{class} \rightarrow \mathrm{dom}.x.\mathrm{avpairs} = \mathrm{dom}\ y.\mathrm{avpairs}$$

B. Conditions

In this section, we shall describe the form of the *condition* part of a production in our system, and show how such conditions may be satisfied by a working memory. In order to maintain consistency between this article and the OPS5 Users Manual, we shall refer to the condition part of a production as a left-hand side (LHS).

Informally, a LHS consists of a collection of *condition elements*, each of which may *match* an attribute-value element in the working memory. An example of a condition element is:

block	color	blue	← a value
	length	*X*	← a variable
	breadth	*X*	

In this example, we introduce a *variable*, *X* (in all examples, single capital letters will denote variables) which matches *any* value. This condition element will match any block whose color attribute has the value blue, and whose length and breadth attributes have the same value; if the length and breadth attributes were matched to different variables, the condition element would match *any* block with the color blue. Note also, that a condition element need not specify

values for all declared attributes of a class–in this case the attribute height is missing.

A LHS is *satisfied* by a working memory if, and only if, each of its condition elements matches an attribute-value element in the working memory (note that several condition elements in a LHS may match one particular attribute-value element in the working memory).

Before we begin to formalize condition elements and LHSs we introduce another given set, VAR—the set of all variable names. Any element of the set VAR may be *bound* in the usual way to a unique element in the set VALUE; a *binding* is a function from variables to values:

BINDING

$$VAR \pfun VALUE$$

We now introduce the idea of a *value specifier,* which may be either a value or a variable, and define the set VSPEC, as the disjoint union of set VAR and VALUE:

VSPEC

$$var \langle\langle VAR \rangle\rangle \mid val \langle\langle VALUE \rangle\rangle$$

The usual injective constructor functions are defined:

$$var : VAR \rightarrowtail VSPEC$$
$$val : VALUE \rightarrowtail VSPEC$$

$$ran\ var \cup ran\ val = VSPEC$$
$$ran\ var \cap ran\ val = \emptyset$$

A condition element consists of a class-name, and some attribute, value-specifier pairs. The class-name and associated attributes must be declared:

CE

$$class : CLASS$$
$$avspecs : ATTR \pfun VSPEC$$

$$class \in dom\ declared$$
$$dom\ avspecs \subseteq declared\ (class)$$

Note that our example condition element from above should now be written slightly differently:

block	color	val (blue)
	length	var (*X*)
	breadth	var (*X*)

A LHS is, then, some set of condition elements (in fact, in OPS5, the LHS is specified as a *sequence* of condition elements for various implementational reasons, but we can ignore this restriction here. Note though, that by making

this simplification we lose the capability of including duplicated condition elements in a LHS):

```
LHS
    condelts : (ℙ CE)
```

A function which will prove useful in *varsin*, which extracts all the variables which appear in a condition element:

```
    varsin : CE → (ℙ VAR)
─────────────────────────────────────
    varsin = λ ce : CE .
                ran (ce.avspecs⨾var⁻¹)   (1)
```

(1) ce.avspecs;var^{-1} is a composition of functions of type ATTR $-|\rightarrow$ VSPEC and VSPEC $-|\rightarrow$ VAR, which yields a function of the type ATTR $-|\rightarrow$ VAR. The domain of this function is the set of attributes in the condition element which map onto variables (as opposed to values). The range of this function gives us the required set of variables.

An *instantiation* of a condition element involves a successful matching of the condition element to an attribute-value element, and the resulting binding of the variables in the condition element to the corresponding values in the matched element, e.g.,

CE				AV-ELT	
block	color	val (red)	matches	block	color red
	length	var (X)		length	2
	height	var (X)		height	4

producing

$$\{ X \mapsto 2, Y \mapsto 4 \}$$

More formally:

```
CE-INST
    CE
    AV-ELT'
    binding : BINDING
─────────────────────────────────────
    class = class'                                  (1)
    dom binding = varsin (θCE)                      (2)
    avspecs⨾val⁻¹ ⊆ avpairs'                        (3)
    avspecs⨾var⁻¹⨾binding ⊆ avpairs'                (4)
```

(1) In order for an attribute-value element to match a condition element, they must have the same class-name.

(2) The resulting binding must be a binding of all the variables in the condition element, and only those variables.

(3) avspecs;val^{-1} is a composition of functions of type ATTR $-|\rightarrow$ VSPEC and VSPEC $-|\rightarrow$ VAL, which produces a function type ATTR $-|\rightarrow$ VAL, so in the above example:

```
    avspecs                          val⁻¹
    -------                          ---
{colour ↦ val (red),          {val (red) ↦ red,
 length ↦ var (X),      ;      val  (4)  ↦ 4,...)  =  {colour ↦ red}
 height ↦ var (Y)}
```

By specifying that this set must be a subset of avpairs, we ensure that each attribute of the condition element which maps onto a value must map onto exactly the same value in a matched attribute-value element.

(4) avspecs;var^{-1}; binding is a composition of functions of type ATTR $-|\rightarrow$ VSPEC, VSPEC $-|\rightarrow$ VAR and VAR $-|\rightarrow$ VALUE, and so is of type ATTR $-|\rightarrow$ VALUE, e.g. above:

```
        avspecs                 var⁻¹                binding
        -------                 ---                  -------
{colour ↦ val (red),      {var (X) ↦ X,            {X ↦ 2,
 length ↦ var (X),    ;    var (Y) ↦ Y,..)   ;      Y ↦ 3}  =
 height ↦ var (Y)}
                  {length ↦ 2, height ↦ 3}
```

By specifying that this set must be a subset of avpairs, we ensure that for each attribute "A" of the condition element which maps onto a variable "V", the following must hold: the value to which "V" is bound in binding must be the value onto which "A" maps in the attribute-value element (e.g., above, length maps onto X in the condition element, and X maps onto the value Z in the binding, which is indeed the value of length in the matched attribute-value element). Note that the functionality of the binding requires that any attributes of the condition element which map onto the same variable must match the same value.

An instantiation of a LHS consists of an instantiation of each of its condition elements with the restriction that any variables which appear in more than one condition element must be consistently bound:

```
LHS-INST ──────────────────────────────────────────────────────
        LHS
        matched : WM
        ceinsts : (P CE-INST)
        lhsbinding : BINDING
        ─────────────
        λCE-INST.θCE (ceinsts) = condelts          (1)
        λCE-INST.θAV-ELT' (ceinsts) = matched      (2)
        lhsbinding = ∪ inst.binding                (3)
                     inst : ceinsts
```

(1) the projection function is used here to extract each of the instantiated condition elements. These are, of course, exactly those which appear in the LHS.

(2) As in (1), the projection function is used, this time to extract each of the matched attribute-value elements. It will prove useful to have explicitly named this collection of matched attribute-value elements. Note that we allow several condition elements to match the same attribute-value element, e.g., suppose a LHS is composed of the following condition elements:

block	color val (red)	block	length var (X) height var (X)

These might both match the following attribute-value element in working memory:

block	color	red
	length	5
	breadth	1
	height	5

In this case the set matched would contain a single element, the attribute-value element above.

(3) The functionality of the lhsbinding ensures that any variables which appear in more than one condition element must match the same value.

C. Actions

We now describe the action part of a production in our system. Again, to maintain consistency with the OPS5 document, we shall rename an action part as a right-hand side (RHS).

A RHS is a collection of operations which specify changes to be made to the working memory. The operations which we shall allow to *make*, and *remove*. The make operation builds a new attribute-value element to be added to the working memory; the element built by a make operation is specified by a pattern which takes the same form as a condition element. We can illustrate how this works informally; given a condition element and binding of the variables which appear in the condition element, e.g.,

```
block   colour  val (red)        {X ↦ 2,Y ↦ 3}
        length  var (X)
        height  var (Y)
```

We can construct a unique corresponding attribute-value element:

block	color	red
	length	2
	breadth	NIL
	height	3

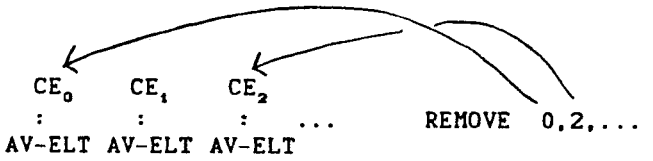
Note that any attributes which are not specified in the condition element are given the default value NIL—this is a special member of the set of VALUEs

which is used in these circumstances. We can define a function which constructs an attribute-value element in this way:

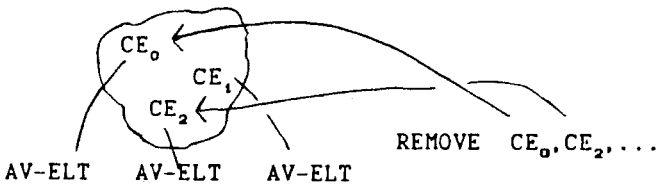
```
make : (CE × BINDING) ⇸ AV-ELT

make =
  λ(ce,bind) : CE × BINDING | varsin (ce) ⊆ dom bind .
       τAV-ELT | class = ce.class ∧ avpairs =
          {att ↦ NIL | att ∈ declared (class)} ⊕      (1)
           ce.avspecs⨾((var⁻¹⨾bind) ∪ val⁻¹)          (2)
```

(1) The attribute-value pairs can then be constructed by giving *all* the declared attributes the default value NIL, and then (2) overwriting with the values specified in the CE together with the binding. The functions are composed (see previously CE-INST) to produce a set of attribute $\mapsto$ value pairs with which we overwrite the attribute $\mapsto$ NIL pairs.

The remove operation specifies the deletion of an element from working memory. The element to be deleted by the remove operation is one of those which has matched a condition element on the LHS. As we mentioned earlier, in OPS5, the LHS is a *sequence* of condition elements; in this case, the element to be removed is indicated by an integer i, which means that the attribute-value element matching the ith condition element in the LHS sequence should be removed from the working memory. This is best illustrated informally:



In order to achieve an analogous effect in our specification, we need some way of indicating a particular condition element on the LHS. The simplest way to do this is by explicitly including the condition element in question:



We can define a function which formalizes this:

```
remove : (LHS-INST × (ℙ CE)) ⇸ (ℙ AV-ELT)

remove =
  λ(lhsinst,setce) : LHS-INST × (ℙ CE) |
                        setce ⊆ lhsinst.condelts .     (1)
       λCE-INST.θAV-ELT'{(ceinst:inst.ceinsts |
                          ceinst.θCE ∈ setce)}          (2)
```

(1) When specifying what must be removed, we should only "refer" to condition elements on the LHS.

(2) We restrict the set of condition element instantiations to those whose condition elements is in the set to be removed, and then extract the matched attribute value elements from this resulting set using the projection function.

A RHS is some collection of these make and remove operations, all of which may be specified, as described above, by conditional elements:

```
RHS
    makes, removes : (ℙ CE)
```

A production consists of a LHS and a RHS:

```
PRODUCTION
    LHS
    RHS
─────────────────────────────────────────
    removes ⊆ condelts                          (1)
    U varsin {makes} ⊆ U varsin {condelts}      (2)
```

(1) All the removes should refer to a condition element on the LHS.

(2) The variables in the condition elements of the makes on the RHS must appear somewhere on the LHS (so that they will be bound when the production has been instantiated).

A production instantiation consists of a production and an instantiation of the LHS of the production:

```
INST
    PRODUCTION
    LHS-INST
```

Clearly, a particular production may be instantiated in several different ways (i.e., the LHS may match several different sets of attribute-value elements) given a particular working memory. We can define a function which produces the set of all possible instantiations of a production in a working memory:

```
allinst : (PRODUCTION × WM) → (ℙ INST)
─────────────────────────────────────────
allinst = λ(p, wmemory) : PRODUCTION × WM .
    (INST | θPRODUCTION = p ∧ matched ⊆ wmemory)
```

The RHS of a production can be carried out only once the production has been instantiated. We can define a function *apply* which illustrates the effect of carrying out the operations specified in a production's RHS, given a particular instantiation of the production from the working memory. Informally, a new working memory is obtained by removing the attribute-value elements specified in the removes part of the RHS, and then adding the newly constructed elements, as specified in the makes part:

```
apply : (INST × WM) ⇸ WM
─────────────────────────────────────────────────────────
apply = λ(inst,wmemory) : INST × WM |
              inst ∈ allinst (inst.θPRODUCTION,wmemory) .
         (wmemory' : WM |
              wmemory' = (wmemory - DELETES) ∪ ADDS
  WHERE
       DELETES = remove (inst.θLHS-INST,inst.removes))   (1)
       ADDS = ∪ { make (ce,inst.lhsbinding) }            (2)
            ce ∈ inst.makes
```

(1) This function call produces the set of attribute-value elements to be removed.

(2) This constructs the set of attribute-value elements to be added to the working memory.

As we would expect, the make operations may exactly cancel out the effect of the remove operations in which cases, apply(inst,wmemory) = wmemory. This happens when DELETES = ADDS, and of course includes the trivial case where inst.makes = inst.removes = {}

At this point, we can begin to see how this specification relates to the more abstract specification presented earlier. Recall that we defined a function *rel*, which given a production, produces the relation (between working memories) which the production "represents". We can now define *rel* for this specification:

```
rel : PRODUCTION → (WM ↔ WM)
─────────────────────────────────────────────────────────
rel = λp : PRODUCTION .
    ((wmemory,wmemory') : WM × WM |
         ∃ INST : allinst (p,wmemory) |
              wmemory' = apply (θINST,wmemory))
```

We can now see why *rel* returns a relation, and not a function. As we have seen, a production may be instantiated in several different ways in a working memory; when applied, each of these instantiations may produce a different resulting working memory, so that for a particular production, a given working memory may map to several different working memories.

As we would expect, the state of the system is defined by a production memory, together with a working memory:

```
STATE
─────────────────────────────────────────────────────────
pmemory : (ℙ PRODUCTION)
wmemory : WM
```

The recognize-act cycle now corresponds closely to that of our previous specification:

RACYCLE

ΔSTATE

∃ p ∈ pmemory | (wmemory, wmemory') ∈ rel (p)

A ΔState, as in the previous specification, includes a predicate stating that the pmemory does not change. Note that, as previously, this specification is nondeterministic, since we do not specify which instantiation of the selected production should be used, when there are several available. The conflict resolution strategy might, in fact, be very simple, e.g., choose any instantiation of any satisfied production, but usually a more complex strategy is proposed.

VII. EXTENDING THE SPECIFICATION

Having created the basic constructional framework of our specification, we shall now add some new facilities, which make it resemble the OPS5 system more closely. More specifically, we shall extend the capabilities of the LHS to allow more powerful pattern matching and negated condition elements. This is necessary for as the specification stands, we would find it difficult to solve even such simple problems as that of the water jugs described earlier. This is because the capabilities of the conditions are too limited; we can not express conditions such as "the 4-gallon jug is not full" in the concise way in which the predicate $X < 4$ does.

A. More Powerful Pattern Matching

In the previous specification, a condition element consisted of a classname, plus some attribute, value-specifier pairs, in which a particular attribute's value was specified in an "all or nothing" fashion, by either a value or a variable. In OPS5 however, more powerful pattern matching facilities are provided, by which the user may specify a *range* of values within which the matched value must lie. Without listing all of the available facilities, we can illustrate the kind of things that can be done, through a few simple examples:

$$\{> 1 < 5\}$$

This pattern will match any value which lies between 2 and 4 inclusive.

$$\{a\ b\ c\ d\ e\}$$

This pattern denotes the disjunction of values, and will match any of the values, *a*, *b*, *c*, *d*, or *e*.

$$\{> 1 <> X\}$$

In this example, we introduce the idea of testing a previously bound variable—this pattern will match any value which is greater than 1 and not equal to the current binding of the variable X.

From these few examples, it should be clear that certain pattern operators

(e.g., <, >) should be applied only to a certain subset of the possible values (the numeric ones), and that if we wanted to specify patterns in detail, we would need to introduce more structure into the set VALUE. We purposely decide to ignore the internal structure of the patterns and the problems of type, and introduce a new set, PATTERN—the set of all possible patterns.

We can see that a pattern may be *evaluated* to yield a set of "allowed" values which it represents; of course, any variables which appear in the pattern must each be bound to a unique value, in order to allow such an evaluation.

It will be useful if we can extract the set of variables tested in a pattern:

```
varsinpatt : PATTERN → (ℙ VAR)
```

For example, varsinpatt ({ > 1 < Y}) = {Y}

The evaluation function, given a pattern and a binding of all the variables in the pattern, will produce the set of "allowed" values which the pattern represents:

```
eval : (PATTERN × BINDING) ⇸ (ℙ VALUE)
-----------------------------------------
dom eval = {(p,b) : PATTERN × BINDING |
                        varsinpatt (p) ⊆ dom b}
```

The application of this function to the above example patterns can be illustrated *informally* as follows:

```
eval ( { > 1  < 5 } , {} ) = {2,3,4}

eval ( { a b c d e } , {} ) = {a,b,c,d,e}

eval ( { > 1  <> X } , {X ↦ 5} = {2,3,4,6,7,...}
```

We want to modify our definition of a condition element, so that the values to be matched are now specified by patterns, e.g.,

block	color	{red blue}
	length	{ > 5 < 10}

However, if we are to allow variables to be *tested* within the patterns, these variables must be bound, elsewhere, prior to testing; with this in mind, we "tag onto" each condition element pattern a variable, to which the matched value (if one is found) will be assigned:

block	color	{red blue}	X
	length	{ > 5 < 10}	Y
	height	{ < Y }	Z

It is important to note the different uses of variables illustrated in this example; the variables X, Y, and Z appearing on the far right are free variables

to which a matching value will be *assigned*. The variable Y appearing in the height pattern $\{ < Y \}$ is used for an entirely different purpose—it is assumed to have been bound to a particular value, which is *tested*. Here, we can begin to see how variables may be tested usefully within patterns—any matching attribute-value element must have a height value less than its length value.

An implementational restriction of OPS5 worth noting is that there is a known order in which the attributes of a condition element are matched (and therefore, in which the variables are bound). This necessitates that variables may only be tested in patterns which occur "after" (in the order of matching) the position in which the variable is bound. So, supposing that, in the above example, the attributes were matched in the order color, length, height, we could test the value of Z in either of the preceding patterns. We shall, for the sake of simplicity, choose to ignore this restriction, and allow variables to be tested irrespective of where in the condition element they are bound.

Recall that in the previous specification, a value-specifier was either a value or a variable. We redefine a value-specifier as an object which consists of a pattern, and a free variable:

```
VSPEC1
    patt     : PATTERN
    assignto : VAR
-------------------------
    assignto ∉ varsinpatt (patt)
```

In OPS5, either the pattern or the free variable may be absent from a value specifier, but we keep the specification simpler by requiring that both be present. This seems a reasonable thing to do, since, we may regard an absent pattern as one which matches any value; implicitly defined variables might easily be introduced into the specification at a later date, to deal with the possibility of absent variables.

Note that we have added on a "1" suffix to denote a modified component; we shall continue to use this convention. Comparing with the capabilities of the previous version of a value-specifier, we can see that the pattern component may be used to specify a single value by using a pattern such as {=red}. To specify an "unrestricted" variable, an empty pattern may be used.

We can now formally define one of these new condition elements which we have described informally above. An important point to note is that each attribute has a *different* free variable to which a matched value will be assigned:

```
CE1
    class : CLASS
    avspecs : ATTR ⇸ VSPEC1
-------------------------
    class ∈ dom declared
    dom avspecs ⊆ declared (class)
    ∀ a1,a2 : dom avspecs . a1 ≠ a2 ⇔
          λVSPEC1.assignto (avspecs (a1)) ≠
          λVSPEC1.assignto (avspecs (a2))        (1)
```

(1) This predicate ensures that each attribute has a different free variable. Note also that our new specification of a condition element permits variables to be tested which are not assigned to *anywhere* in the condition element. (We shall see the reason for this later).

In order for an attribute-value element to match one of these new condition elements, their class names must be the same, and each matched attribute's value must fall within the range of values specified by the corresponding pattern in the condition element, e.g.,

		CE1					
block	color	{ <> blue }	X	matches	block	color	red
	length	{ > 3 }	*Y*			length	5
	height	{ <> Y}	*Z*			breadth	6
						height	7

because

```
block = block
red ∈ eval (( <> blue ),binding)
5  ∈ eval (( > 3 ),binding)
7  ∈ eval (( <> Y ),binding)

    WHERE binding = (X↦red, Y↦5, Z↦7)
```

Before formalizing these ideas, we introduce two more functions, *assigned*, which extracts all the variables which are *assigned* to in a condition element, and *tested*, which extracts all of the variables *tested* in the patterns of a condition element:

```
assigned : CE1 → (P VAR)
tested   : CE1 → (P VAR)
─────────────────────────
assigned =
  λce : CE1 . λVSPEC1.assignto (ran ce.avspecs)
tested  =
  λce : CE1 . varsinpatt ( λVSPEC1.patt (ran ce.avspecs) )
```

So, applied to the condition element in the above example, assigned will produce the set $\{X, Y, Z\}$ and tested will produce the set $\{Y\}$.

As before, an instantiation of a condition element consists of the condition element, a matching attribute-value element and the resulting binding of variables (remember that some of the tested variables may not be assigned to in the condition element—these variables are assumed to have been bound elsewhere in the LHS instantiation of which the condition element instantiation is part):

```
CE-INST1 ──────────────────────────────
    CE1
    AV-ELT'
    binding : BINDING
```

```
class' = class                                                    (1)
dom binding = assigned (θCE1) ∪ tested (θCE1)                     (2)
a ∈ dom avspecs ⟹
  avpairs' (a) ∈
     eval (λVSPEC1.patt (avspecs(a)),binding) ∧                   (3)
  avpairs' (a) =
     binding (λVSPEC1.assignto (avspecs(a)))                      (4)
```

(1) In order for an attribute-value element to match a CE1, they must have the same class-name.

(2) The binding must be a binding of all the variables in the condition element and only those variables.

(3) Each matched attribute's value must fall within the set of values allowed by the corresponding condition element pattern.

(4) The binding must bind each free variable in the condition element to the "correct" value, i.e., the value of the attribute with which it is associated in the condition element. For example, in the above example, X must be bound to red (and not 5, 6, or 7).

B. Negated Condition Elements

Until now, we have regarded a condition element as an object which specifies something which must be *present* in the working memory; a LHS was a collection of condition elements, each of which had to match something in the working memory in order for the LHS to be satisfied. In OPS5, however, condition elements may be *negated;* a negated condition element is simply a condition element with an indicator attached which implies that the condition element should *not* match anything in the working memory.

A LHS is now satisfied by a working memory if each of its positive (i.e., nonegated) condition elements can be successfully instantiated with an element in the working memory, and none of its negated condition elements can be instantiated given the binding produced from the instantiations of the positive condition elements.

We can begin to formalize these ideas by introducing a new set of condition elements which is composed of the disjoint union of the sets of negated and positive condition elements:

```
CE2
        neg <<CE1>> | pos <<CE1>>
```

The usual constructor functions are assumed to be defined. A LHS consists, as before, of a collection of condition elements, but now, any of these may be negated (in fact, in OPS5 the sequence of condition elements must begin with at least one positive condition element, but we shall choose to ignore this restriction):

```
LHS1
    condelts : (ℙ CE2)
─────────────────────────────────────────────────────────────
    tested (ce1s) ⊆ assigned (pos⁻¹(condelts))                    (1)
    ∀ ce,ce' : ce1s . ce ≠ ce' ⇒
                    assigned (ce) ∩ assigned (ce') = ∅            (2)
    WHERE ce1s = (pos⁻¹(condelts) ∪ neg⁻¹(condelts))
```

(1) Any variables which are tested in a LHS must be "assigned to" in some *positive* condition element of the LHS (since the free variables appearing in negated condition elements will *not* be assigned to in a "successful" matching).

(2) Each variable is assigned to only once, in a LHS.

As we discussed earlier, the order in which the matching is carried out is important since a pattern can not be evaluated until all of its tested variables have been bound. In OPS5, the sequence of condition elements is matched in "left to right" order, so that variables may only be tested once they have been bound "somewhere to their left" in the sequence. In choosing to represent the LHS as an unordered collection of condition elements, we have purposely ignored such implementation restrictions, thereby keeping the specification simpler.

Two functions which will be useful are *positives*, which extracts all the positive condition elements from a LHS1, and *negatives*, which extracts all the negated condition elements:

```
    positives : LHS1 → (ℙ CE1)
    negatives : LHS1 → (ℙ CE1)
─────────────────────────────────
    positives =
      λlhs : LHS1 . pos⁻¹(lhs.condelts)
    negatives =
      λlhs : LHS1 . neg⁻¹(lhs.condelts)
```

An instantiation of a LHS now consists of an instantiation on each of its positive condition elements—we do not include here the idea that the negated condition elements must not match anything in the working memory (we formalized this later, in the definition of the *allinst* function):

```
LHS-INST1
    LHS1
    matched : WM
    ceinsts : (ℙ CE-INST1)
    lhsbinding : BINDING
─────────────────────────────────────────────────────────────
    λCE-INST1.θCE1 (ceinsts) = positives (θLHS1)    (1)
    λCE-INST1.θAV-ELT' (ceinsts) = matched          (2)
    lhsbinding = ⋃  inst.binding                    (3)
                 inst : ceinsts
```

(1) The condition elements instantiated are the positive condition elements in the LHS.

(2) The matched attribute-value elements are those which appear in the condition element instantiations.

(3) The functionality of the binding ensures that all variables must be consistently bound.

C. Resulting Changes in the Specification

We shall not extend the capabilities of an action part in our system—in OPS5, a *modify* command is provided, by which matched attribute-value elements may be modified, but, since an equivalent effect can be achieved using the make and remove commands, this is somewhat superfluous. In addition, there are actions which fall outside those permitted in a pure production system, e.g., I/O operations, actions which add productions to the production memory, and so on. Here, however, we shall simply note the changes which our extensions to the LSH necessitate.

As we have said, we shall not extend the capabilities of an action part. The specification of a RHS must be modified so that the removes have the correct type:

```
RHS1
        makes : (ℙ CE)
        removes : (ℙ CE1)
```

A production is now:

```
PRODUCTION1
        LHS1
        RHS1
        ─────
        removes ⊆ positives (θLHS1)                          (1)
        ∪ varsin (makes) ⊆ ∪ assigned (positives (θLHS1))    (2)
```

(1) The removes should only refer to positive condition elements, since the negative condition elements will *not* match anything in a successful instantiation.

(2) The variables appearing on the RHS in the elements to be built must be assigned to in some positive condition element in the LHS (since only these variables will be bounded). A production instantiation looks much as it did previously:

```
INST1
        PRODUCTION1
        LHS-INST1
```

We can now describe the intended effect of the negated condition elements more formally. The *allinst* function which produces the set of all instantiations of a production in a particular working memory must be modified to take into

account the fact that an instantiation of each of the positive condition elements in the LHS represent a valid instantiation of a production only if each of the negated condition elements *cannot* be instantiated given the resulting binding:

```
allinst1 : (PRODUCTION1 × WM) → (ℙ INST1)

allinst1 = λ(p,wmemory) : PRODUCTION1 × WM .
        {INST1 | θPRODUCTION1 = p ∧
                 matched ⊆ wmemory ∧                    (1)
                 ∀ ce : negatives (θLHS1) .
                     ~ ∃ CE-INST1 |
                           θCE1 = ce ∧
                           θAV-ELT' ∈ wmemory ∧
                           binding ⊆ lhsbinding}        (2)
```

(1) All of the matched elements must be from the working memory.

(2) This predicate ensures that, as we stated above, it is not possible to instantiate any of the negated condition elements with an element from the working memory, given the binding produced as a result of instantiating the positive condition elements.

The effect of carrying out the operations of a RHS is, informally, exactly the same as in the previous specification—the new working memory is obtained by removing and adding attribute-value elements, as specified. We shall not detail here the remaining changes required, since they are straightforward, and can, in fact involve simply suffixing with ones where appropriate. As one example, we see that the new *rel* function is:

```
rel1 : PRODUCTION1 → (WM ↔ WM)

rel1 = λp : PRODUCTION1 .
    {wmemory,wmemory': WM × WM |
          ∃ INST1 : allinst1 (p,wmemory) |
              wmemory' = apply1 (θINST1,wmemory)}
```

This definition is, in fact, of exactly the same form that of *rel*, the only difference being that certain components now have "1" suffixes.

An exception to this claim that new components can be derived by simply adding suffixes, is the *remove* function, which must be modified to take into

```
remove1 : (LHS-INST1 × (ℙ CE1)) ⇸ (ℙ AV-ELT)

remove1 =
  λ(inst,setce) : LHS-INST1 × (ℙ CE1) |
          setce ⊆ positives (inst.θLHS1) .
     λCE-INST1.θAV-ELT' {c:inst.ceinsts |
                              c.θCE1 ∈ setce}
```

account the possibility of negated condition elements in a LHS instantiation (clearly, we should only "point to" positive condition elements in the LHS when specifying what is to be removed).

VIII. GOALS REVISITED

We now return to the question of *goals*. Recall the water jug problem, in which the goal was to leave 2 gallons of water in the 4-gallon jug. Suppose that we had introduced an attribute-value element to represent each empty jug:

jug	capacity	4	jug	capacity	3
	contents	0		contents	0

We might then represent the goal by a condition element:

jug	capacity	{=2} *X*
	contents	{=2} *Y*

Informally, then, we can see the predicate which tests whether the goal is satisfied will be of the form:

∃ something in the working memory which
instantiates the above condition element

More generally, a goal might be represented by a *collection* of condition elements, some or all of which might be negated. Thus, we may represent a goal by a LHS. Informally, a goal is satisfied if we can produce an instantiation of this goal given the current contents of the working memory. As we would expect, the formal definition looks somewhat similar to that of the *allinst1* function:

```
GOAL-SAT
        goal : LHS1
        wmemory : WM
-------------------------------------------------------------
        ∃ lhsinst : LHS-INST1 |
            goal = lhsinst.θLHS1 ∧
            lhsinst.matched ⊆ wmemory ∧
            ∀ ce : negatives (lhsinst.θLHS1) |
                ~ ∃ ceinst : CE-INST1 |
                            ceinst.θCE1 = ce ∧
                            ceinst.θAV-ELT' ∈ wmemory ∧
                            ceinst.binding ⊆ lhsinst.lhsbinding
```

We can now specify a recognize-act cycle which includes a goal:

RACYCYL-GOAL = RACYCLE & (~ GOAL-SAT)

IX. CONFLICT RESOLUTION

We now consider the question of conflict resolution, which has been ignored in our previous specifications. Either, we stated that the set of productions satisfied by the working memory on any one recognize-act cycle is known as the *conflict set* and that the task of the conflict resolution strategy is to select one production from this set to be applied. We have seen that in our system, a production may be instantiated in several different ways given a particular working memory, so that the task of conflict resolution is somewhat more complicated—we must select one *instantiation* of one production. For this reason, when we refer to the *conflict set,* we shall mean the set of production *instantiations* from which we may choose on any particular cycle.

We shall extend the specification of RACYCLE1 from the previous section to include a conflict resolution strategy which is based on the LEX strategy available in OPS5 (goals are not relevant to this discussion of conflict resolution, and are therefore omitted). Before beginning, we define a new term; we have seen that a particular production instantiation may be applied (i.e., we may carry out the actions on the RHS) to produce a new working memory. When an instantiation is applied, we shall say that this instantiation has been *fired.*

A. Time Tagging

Our conflict resolution strategy shall make use of the *time* at which at attribute-value element is added to the working memory. More specifically, we shall give preference to instantiations matching elements which have been most recently added to the working memory (we shall discuss the reason for this later). Clearly, we must have some method of discerning the relative times at which elements have been added to the working memory. This is achieved in OPS5 by tagging each element with a numeric *time tag* as it is added to the working memory, the larger the time tag, the more recent the element.

To formalize this, we introduce a new object, a *working memory element,* (WME), consisting of an attribute-value element and a time tag:

```
WME
    AV-ELT'
    tag : N
```

The set of all working memories is now:

```
WM1
    P WME
```

Note that this permits duplicated attribute-value elements to be present in the working memory (with different time tags).

Condition elements are now instantiated with WMEs; in order for a condition element to match a WME, it must match (as described previously) the attribute-value element component on the WME:

```
CE-INST2
    CE-INST1
    WME
```

Note that AV-ELT' is a component of both CE-INST1, and WME, and will therefore be identified, so that we have neatly specified the new condition element instantiation.

We shall not describe the resulting changes to the specification here, since they are fairly straightforward; the important point to note is that when a new WME is added to the working memory, it is given a time tag greater than those of the elements currently in the working memory. This is described in a modified apply function:

```
apply2 : (INST2 × WM1) ⇸ WM1

apply2 =
  λ(inst,wmemory) : INST2 × WM1 |
        inst ∈ allinst2 (inst.θPRODUCTION1,wmemory) .
    μ(wmemory' : WM1 | wmemory' =
          (wmemory - DELETES2) ∪ NEW-WMES)
WHERE
     DELETES2 = remove2 (inst.θLHS-INST2,inst.removes)
     NEW-WMES = ∪ newtag (av-elt,wmemory)                    (1)
                av-elt : ADDS2
     ADDS2 = ∪ (make (ce,inst.lhsbinding))                   (2)
             ce : inst.makes
```

(1) The new WMEs are constructed by (2) first constructing attribute-value elements in the usual way, and then adding a "more recent," i.e., larger time tag than currently exists in the working memory, to each. Note that the ordering of tags among these new elements is not defined.

The function which assigns new time tags is defined as follows:

```
newtag : (AV-ELT × WM1) → WME

newtag = λ (av-elt,wmemory) : AV-ELT × WM1 .
            μ(WME | θAV-ELT' = av-elt ∧
                    θWME.tag > max (wmemory).tag)
```

We shall assume that our specification has been modified and that all the required new components have been defined with suffixes suitably increased. The resulting recognize-act cycle corresponding to RACYCLE1 (i.e., with no conflict resolution) is:

```
RACYCLE2
    ΔSTATE2

    ∃ p : pmemory | (wmemory,wmemory') ∈ rel2 (p)
```

The conflict resolution strategy will consist of two rules which are applied in order. The first rule eliminates certain instantiation from the conflict set, and the second rule uses the time tags to select one of the remaining instantiations.

X. CONFLICT RESOLUTION RULE 1

The first rule may be described as follows:

1. Discard from the conflict set any instantiations that have already fired on a previous cycle. If all the instantiations have already fired, conflict resolution fails and the interpreter halts.

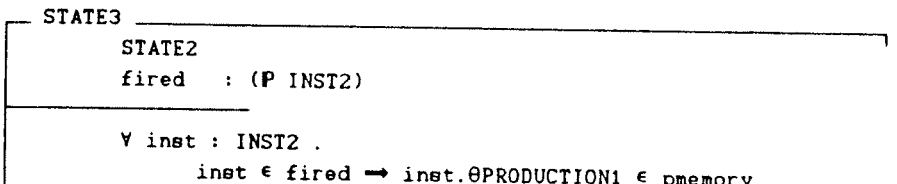
The motivation for this rule is quite clear. Suppose we have a production which might be represented informally as *if there is a red block in the working memory then add a blue block to the working memory*. This production, once instantiated, might be repeatedly selected and fired, forever adding more blue blocks to the working memory. It is unlikely that this is the intended effect of the production—in fact, we would probably only want it to fire once, as this rule ensures.

In order to know if an instantiation has already fired, we must be able to compare two instantiations and decide if they are the same. We shall assume that two instantiations are the same if the are instantiations of the same production, and they have matched the same working memory elements. Two working memory elements are the same if they are composed of the same attribute-value element and time tag.

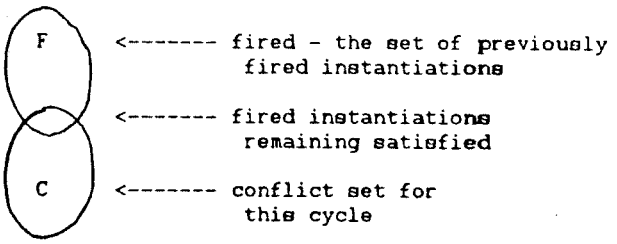
There is, however, one rather obscure exception to this rule in OPS5, which may best be illustrated by an example. Suppose we have a production of the following form: *if there is a red block in the working memory and there is not a blue block in the working memory then* . . . , and that this production is instantiated with a particular red block, B, say, when there are no blue blocks in the working memory. We fire the instantiation once, and, although it remains satisfied, we do not fire the instantiation again (because of rule 1).

Suppose however, that some time later, a blue block enters the working memory, which causes the instantiation to become unsatisified. If this blue block is subsequently deleted, then, assuming that red block B is still in the working memory, the original instantiation will "reappear". Although this instantiation is *equal* (as described above) to the one which we fired earlier, we *do* want to allow the instantiation to fire again, because its relevance has been, in some way, *reestablished* by its reappearance.

Fortunately, these somewhat complicated sounding ideas can be formalized fairly easily, and in a manner which aids understanding. We can begin by introducing a new component to the state, which represents the set of instantiations which have fired "so far":



We can now describe diagrammatically the *recognize* part of any particular recognize-act cycle:



To follow the course of actions prescribed by rule 1, we should obviously fire some instantiation from the set $C - F$. The exception noted above may now be formalized quite easily. Any instantiations which have been fired, and which remain satisfied, will reside in the set composed of the intersection of C and F—these are the instantiations which we do not want to fire again. The instantiations in the set $F - C$ have become unsatisfied, and should be allowed to refire, if they reappear. We can permit this by simply removing all such instantiations from the set F. Thus, after each cycle, the set F should be modified so that it contains only those instantiations in the set $F \cap C$.

A recognize-act cycle which includes rule 1 is:

```
RACYCLE3 ____________________________________________________
        ΔSTATE3
        INST2
        ------------------------------------------------------
        θINST2 ∈ conflictset - fired                    (1)
        wmemory' = apply2 (θINST2,wmemory)
        fired' = ( fired ∩ conflictset ) ∪ (θINST2)     (2)
            WHERE  conflictset = ∪ allinst2 (p,wmemory)
                                 p : pmemory
```

(1) As we noted above, the instantiation acted upon should be one which has not already fired.

(2) This predicate ensures that only those instantiations which have fired, *and* which remain satisfied, will not be fired again. Note that we must include the instantiation which we fire on this cycle.

XI. CONFLICT RESOLUTION RULE 2

The second rule may be stated as follows.

2. Order the remaining (after rule 1 has been applied) instantiations on the basis of the recency of the working memory elements, using the following algorithm to compare pairs of instantiations: first compare the most recent elements from the two instantiations (i.e., those with the largest time tags). If one element is more recent than the other, the instantiation containing that element dominates. If the two elements are equally recent, compare the second most recent elements from the instantiations. Continue in this manner either

until one element of one instantiation is found to be more recent than the corresponding element in the other instantiation, or until no elements remain for one instantiation. If one instantiation is exhausted before the other, the instantiation not exhausted dominates; if the two instantiations are exhausted at the same time, neither dominates.

This rule intended to *focus* the attention of the production system onto a subtask, once this subtask has been started. To see how this works, suppose that some new elements have just been added to the working memory, which constitute the first steps on this subtask. Rule 2 ensures that any instantiations matching these particular elements will be selected. Hopefully, these instantiations will be relevant to the completion of the subtask.

Having already introduced time tags, this rule may be formalized quite easily:

```
RACYCLE4
        RACYCLE3
        ~ ∃ INST2': recognised - fired' |
                    (θINST2') morerecent (θINST2)    (1)
```

(1) There should not be an instantiation which is more recent (as defined informally in rule 2, and more formally below) than the one which is selected to be fired.

The following relation and function formalize the algorithm described in rule 2:

```
morerecent : INST2 ↔ INST2

∀ inst,inst' : INST2 .
  inst morerecent inst' ⇔
            lexico (inst'.matched) < lexico (inst.matched)
WHERE
       < denotes lexicographic ordering on
         sequences of numbers
```

```
lexico : (ℙ WME) → seq [N]

∀ s1 : (ℙ WME) .
    if s1 = ∅ then <>
        else maxim^ lexico (s1 - {melem})
WHERE
     maxim = max {s1.tag}
     melem ∈ {wme : WME | wme.tag = maxim}
```

It is important to note that RACYCLE4 is still nondeterministic—the conflict resolution strategy we have introduced merely serves *to narrow* (although

hopefully, quite considerably) the choice of instantiations from which we must choose on any particular cycle.

XII. CONCLUSIONS

We feel that we have successfully formalized many important aspects of the OPS5 production system framework, and that the specification includes the workings of the system far more clearly than the user documentation. An interesting point to note is that much of the work went into producing the very abstract specification, and that once this model has been produced, it was fairly straightforward to develop the specification of OPS5 corresponding to this model. It should be noted, however, that several aspects of OPS5 were not dealt with; we list some of these here for completeness:

1. We have specified the working memory as a collection of time-tagged attribute-value elements; in fact, the working memory may contain other structures called *vector elements,* which, in terms of our specification, are sequences of values. As with attribute-value elements, these are matched against the condition elements of a LHS. The condition elements are therefore somewhat more flexible (and complicated) then we described, allowing them to match either type of structure. As one would expect, the *make* operation can be used to construct vector elements, as well as attribute-value elements, to be added to the working memory.

2. As we mentioned briefly before, in the section on "extensions to the specification," the RHS operations in OPS5 include several actions not permitted in a "pure" production system, e.g., I/O operations, file operations and calls to user subroutines. Of particular note though, are the BUILD action, by which new productions may be added to the working memory, and the HALT action, which halts the interpreter. We noted that a goal might be represented by a LHS in OPS5; we can see that one may write a "halting production," whose only purpose is to halt the production system when a goal has been reached. The LHS of such a production consists of the goal, the RHS consists solely of a HALT action.

3. In OPS5, the user may specify, when starting the system, the number of cycles the system should execute. If possible, the system executes the prescribed number of cycles, and then halts. When the system is not executing, the user may examine, and, more importantly change the contents of the working memory.

4. Our specification describes the class of production systems which adopt an irrevocable[10] control strategy in which the action of productions can not be undone. While this control strategy is sufficient for certain problems, in many applications, a backtracking facility is essential.

Given the proven utility of OPS5 (recall that R1 is implemented in OPS5), we feel that our specification demonstrates some of the advantages to be gained from a more formal approach to the design and implementation of expert systems (in this case clarity of description). The use of a formalist approach will provide a firmer basis from which all aspects of knowledge-based system devel-

opment can proceed, allowing systems engineers to build upon the success of previous implementations and adapt specifications to meet alternative needs while raising the levels of system correctness and reliability.

References

1. C.C. Morgan, *Programming from Specifications,* Prentice-Hall, Englewood Cliffs, NJ, 1989.
2. B.A. Sufrin and J. He, "Specification, analysis and refinement of interactive processes" in, *Formal Methods in Human–Computer Interaction,* M. Harrison and H. Thimbleby, Eds., Cambridge University Press, Cambridge, England, pp. 153–200.
3. E. Rich, *Artificial Intelligence,* McGraw-Hill, New York, 1983.
4. J.M. Spivey, *Understanding Z., A Specification Language and its Semantics,* Cambridge University Press, Cambridge, England, 1989.
5. J.M. Spivey, *The Z Notation,* Englewood Cliffs, NJ, Prentice-Hall, 1990.
6. I. Hayes, *Specification Case Studies,* Technical Monograph PRG-46, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1985.
7. C.L. Forgy, *OPS5 User's Manual,* Technical Report, CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 1981.
8. P.D. Sherman and J.C. Martin, *An OPS5 Primer: An Introduction to Rule-based Expert Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1990.
9. J. McDermott, "R1: An expert in the computer systems domain," in *Proceedings AAAI-80,* Stanford, CA, 1980.
10. N.J. Nilsson, *Principles of Artificial Intelligence,* Springer-Verlag, Berlin, 1982.