SOPS: STRUCTURED OPS5-BASED PRODUCTION SYSTEM

VILAS WUWONGSE and EKAWIT NANTAJEEWARAWAT
Division of Computer Science
Asian Institute of Technology
P.O.Box 2754
Bangkok 10501
Thailand

ABSTRACT

The main drawback of current production systems is identified as the lack of structured knowledge representation. This paper presents an approach for developing a structured production system, which is based on partitioning rules into a hierarchy of groups according to the concepts they produce and then analyzing the patterns of working memory elements used and produced by the resulting groups in order to determine their dependencies. The approach leads to a methodology similar to those used in software engineering which can make a production system easier to maintain and verify. Based on this methodology, SOPS (Structured OPS5-based Production System), a new production-system programming language and environment has been designed and implemented.

INTRODUCTION

Production systems, also known as rule-based systems, are widely used in artificial intelligence for modelling intelligent behaviour and building expert systems. They have been used extensively to develop large expert systems spanning a variety of applications in the areas including medical diagnosis, automatic configuration of computers, and cognitive modelling of human memory.

Maintaining the correctness, consistency and completeness of the knowledge base in a large production system is a considerable problem. Production systems built with typical current technology are difficult to change once they are built. A significant increase in the number of production rules makes the system difficult to manage. The main reason is that knowledge representations in production systems are not structured enough.
Knowledge is represented in a production system as a collection of rules. The if-then rules have the advantage that they are in a declarative form. However, the rules are expressed in a relatively independent manner and their dependencies is implicitly expressed and not obvious. The rule base of a conventional production system is simply a linear list of such independent rules. The database of facts is the only medium for interactions and dependencies among the rules.

The clarity of a production system decreases dramatically as the system has more rules. This is because the knowledge cannot be modularized and the interactions among rules and the database of facts are complex. When the database is modified by the firing of a particular rule, it is difficult to locate the effects on the whole system. Therefore, such knowledge representations are difficult to develop and maintain, especially for a large knowledge base. To overcome this drawback, structured knowledge-representation methods are needed.

The main objective of this research work is to provide an approach for developing structured production systems and design a new production-system programming language and programming environment that supports the proposed approach. The new environment should provide facilities for analyzing the interactions among the rules in the system.

**BASIC CONCEPTS**

A production system is mainly composed of a set of rules, called rule base or production memory, and a set of facts or current world states, called working memory. A working memory element is a representation of a single object. It consists of a set of attributes the values of which together represent the state of an object at a particular time. Each attribute value exhibits one characteristic of an object. In other words, a working memory element reveals the existence of an object while its attribute values describe the object in detail.

A particular working memory element could be viewed in different ways, depending on properties of interest. Each view corresponds to one concept. In general, working memory elements which have similar properties represent the same concept. For example, each of the following working memory elements:

(Person ^name Nuch ^profession student ^nationality Thai)
(Person ^name Nat ^profession student ^nationality Thai)
represents the concepts “a student”, “a Thai citizen”, “an Asian”, etc. Concepts are often aggregations of other concepts. Some working memory elements may together represent a concept. For example, the following working memory elements:

(Person ^name Pol ^age 40 ^sex male ^profession police)
(Person ^name Ladda ^age 37 ^sex female ^profession nurse)
(Person ^name Nuch ^age 10 ^father Pol ^mother Ladda)

may together represent the concept “a family”.

Concepts could be organized into a classification hierarchy. That is, a concept can be regarded as a generalization of other concepts. For example, “an Asian” is a generalization of “a Thai citizen”. Working memory elements may be classified by means of different and independent properties leading to several hierarchies of concepts. For example, the working memory elements of class person could be classified according to the attribute profession into one hierarchy and according to the attribute nationality into another hierarchy.

**PATTERNS OF WORKING MEMORY ELEMENTS**

An element pattern is a notation for referring to working memory elements. It consists of a class specification followed by a sequence of zero or more attribute specifications. The class specification specifies the element class while attribute specifications specify the restrictions on attribute values. For example, the element pattern (rectangle ^area > 100 cm²) refers to any working memory element of class rectangle the area of which is greater than 100 cm². The element pattern which does not contain any attribute specification places no restriction on any attribute. For example, the element pattern (rectangle) refers to any working memory element of the class rectangle. Special variables denoted by *, **, *** etc. referred to as generic variables, are used in element patterns to specify restrictions on the relations between the values of pairs of attributes. Any working memory element of class rectangle, having the value of attribute breadth equal to the value of attribute length, can be referred to by the element pattern (rectangle ^breadth * ^length *). It should be noted that working memory elements of different patterns may represent the same concept.

[Definition 1: Attribute Specification] An attribute specification is of the form ^A r K, where A is an attribute of a working memory element, r ∈ {=,<,>,<,>,≥}, and K is a constant or a generic variable. The predicate operator r need not be explicitly specified when r is ‘=’. 
[Definition 2: Element Pattern] Let $CL$ be a class name and $SA_1, SA_2, ..., SA_n$ be specifications of the attributes $A_1, A_2, ..., A_n$, respectively. An element pattern $P$ is defined as follows:

1) $(CL)$ is an element pattern which refers to any working memory element of class $CL$.

2) $(CL \ SA_1 \ SA_2 \ ... \ SA_n)$ is an element pattern which refers to some working memory elements of class $CL$ whose attributes $A_1, A_2, ..., A_n$ are specified by $SA_1, SA_2, ..., SA_n$, respectively.

$CL$ is called the class specification of $P$.

There are two types of interpretations of attribute specifications. Consider, for example, the element pattern (rectangle '$breadth' * '$length' * '$area > 100 cm^2$) which refers to any working memory element representing a square whose area is greater than 100 cm$^2$. The attribute specification '$area > 100 \ cm^2$ is interpreted as restricting the value of the attribute area to a member of the set $\{ v \mid v > 100 \ cm^2 \}$, while the attribute specifications '$breadth'$ and '$length'$ are interpreted as restricting the value of the attribute breadth to the value of the attribute length. To refer to these two types of interpretations of attribute specifications separately, the terms independent and relative attribute specifications are defined.

[Definition 3: Independent Attribute Specification] Independent attribute specification is an attribute specification $^A r c$ where $c$ is a constant. The set of possible values (SP) that $A$ could take is given as $\{ v \mid v r c \}$.

[Definition 4: Relative Attribute Specification] Relative attribute specification is an attribute specification $^A r v$, where $v$ is a generic variable.

Let $^A_1 r_1 v, ^A_2 r_2 v, ..., ^A_n r_n v$ be a conjunction of relative attribute specifications and one of the relations, $r_i$, is '$=1$', where $1 \leq i \leq n$, then the semantics of the conjunction is given by $^A_1 r_1 ^A_2 r_2 ^A_3 r_3 ..., ^A_n r_n ^A_i$.

The following three relations between element patterns are introduced to form a basis for presenting the methodology in the subsequent sections.

[Definition 5: Equivalent Element Patterns] Element patterns $P$ and $Q$ are equivalent if and only all the following conditions are satisfied.

1) $P$ and $Q$ have the same class specification.
2) For each independent attribute specification $\land A_r c_i$ in $P (Q)$, there exists an independent attribute specification $\land A_r c_i$ in $Q (P)$ such that the set of possible values $SP_i$ of $\land A_r c_i$ is equal to the set of possible values $SP_j$ of $\land A_r c_i$.

3) For each pair of relative attribute specifications $\land A_k r_k v_i$ and $\land A_i r_i v_i$ in $P (Q)$, if $r_k$ is $'= '$, there exists a pair of relative attribute specifications $\land A_k r_m v_j$ and $\land A_i r_n v_j$ in $Q (P)$ such that $r_m$ is $'= '$ and $r_i$ is the same as $r_n$.

[Definition 6: Disjoint Element Patterns] Element patterns $P$ and $Q$ are disjoint if and only if any of the following conditions is satisfied.

1) $P$ and $Q$ have different class specifications.

2) There exists a pair of independent attribute specifications $\land A_i r_k c_i$ in $P$ and $\land A_i r_j c_j$ in $Q$ such that the intersection of the set of possible values $SP_i$ of $\land A_i r_k c_i$ and the set of possible values $SP_j$ of $\land A_i r_j c_j$ is empty.

3) There exist pairs of relative attribute specifications $\land A_k r_k v_i$, $\land A_i r_i v_i$ in $P$ and $\land A_i r_m v_j$, $\land A_i r_n v_j$ in $Q$ such that both $r_k$ and $r_m$ are $'= '$ and the intersection of the sets $\{ v \mid v r_i c \}$ and $\{ v \mid v r_n c \}$ is empty, where $c$ is a constant.

[Definition 7: Partial Covering Relation] Element pattern $P$ partially covers element pattern $Q$ if and only if all the following conditions are satisfied.

1) $P$ and $Q$ have the same class specification.

2) For each independent attribute specification $\land A_i r_k c_i$ in $Q$, there exists an independent attribute specification $\land A_i r_j c_j$ in $P$ such that the intersection of the set of possible values $SP_i$ of $\land A_i r_k c_i$ and the set of possible values $SP_j$ of $\land A_i r_j c_j$ is not empty.

3) For each pair of relative attribute specifications $\land A_k r_k v_i$ and $\land A_i r_i v_i$ in $Q$, if $r_k$ is $'= '$, there exists a pair of relative attribute specifications $\land A_k r_m v_j$ and $\land A_i r_n v_j$ in $P$ such that $r_m$ is $'= '$ and the intersection of the sets $\{ v \mid v r_i c \}$ and $\{ v \mid v r_n c \}$ is not empty, where $c$ is a constant.

THE PROPOSED METHODOLOGY

This section introduces a new methodology for developing structured OPSS-based production systems. The methodology may be used from the start of the development of a production system or may be applied later in the development process. The following steps are taken in applying the proposed methodology.
Identify control elements

Working memory elements can be classified into two types based on the information they carry. The working memory elements of the first type carry information about the physical objects in the problem domain while those of the second type about conceptual objects such as states and goals. The latter are called control elements. Control elements are used exclusively to signal the states of the computation or to direct the sequence in which problem-solving steps are carried out.

[Definition 8: Control Element] Working memory elements which hold information about conceptual objects such as goals (the aims of problem solving), or states of the computation are called control elements.

In OPS5, control elements are not separated by the architecture. Working memory elements of all different classes are stored together. The objective of this step is to distinguish the working memory elements which are used solely for control purpose from those that carry real data between rules.

Divide the rules into contexts

Rules should be partitioned into groups, called contexts, so that rules in each context in concert perform a well-defined task. During execution, only one context is activated at a time. The active context is determined by the contents of control elements.

The condition part of a rule contains a set of patterns, called condition elements, that enable the rule to fire when all are matched. The condition elements which are to be matched against control elements are termed control condition elements. Rules in the same context have control condition elements of the same pattern. That is, rules can be easily divided into contexts by considering the patterns of their control condition elements.

Two rules are context equivalent if every control element which satisfies the control condition element of one rule will also satisfy the control condition element of the other rule. This is formalized by the following two definitions.
[Definition 9: Use (Element Patterns)] Rule $R$ uses a working memory element of pattern $P$, if and only if any of the following conditions is satisfied.

1) A working memory element of the pattern $P$ can match a condition element of $R$ (denoted as 'uses $P$').

2) A working memory element of the pattern $P$ can match a negated condition element of $R$ (denoted as 'R uses $-P$').

[Definition 10: Context Equivalent] Two rules $R$ and $S$ are context equivalent (denoted by $R_{\text{context}} \equiv S$) if and only if, if $R(S)$ uses a control element of pattern $P$ then $S(R)$ uses a control element of pattern $Q$ such that $P$ and $Q$ are equivalent element patterns.

A context is formally defined as a group of rules which are all context equivalent. The rules that are context equivalent will be grouped together in this step.

[Definition 11: Context] A group of rules, $G$, is called a context if and only if for any rules $R$ and $S$ in $G$, $R_{\text{context}} \equiv S$.

Divide the rules in each context into groups

An obvious way of grouping two rules together is to check whether the patterns of working memory elements they produce are equivalent. But a more meaningful and useful approach is to group the rules based on the concepts they produce. Rules in each context should be divided into groups so that rules in a particular group, when fired, result in the same concept. To do this, the notion of concept equivalence is required, which is formally defined below.

[Definition 12: Produce (Element Patterns)] Rule $R$ produces a working memory element of pattern $P$, if and only if the firing of $R$ satisfies any of the following conditions.

1) Adds a working memory element of the pattern $P$ to the working memory (denoted as 'R produces $+P$').

2) Changes some attribute values of an existing working memory element so that it is of the pattern $P$ (denoted as 'R produces $+P$').

3) Removes the working memory element of the pattern $P$ from the working memory (denoted as 'R produces $-P$').
**Definition 13: Concept Equivalent** Two rules $R$ and $S$ are concept equivalent (denoted by $R_{\text{concept}} \equiv S$) with respect to concept $C$ if and only if $R$ and $S$ produce working memory elements of patterns $P$ and $Q$, respectively, such that both working memory elements of the pattern $P$ and working memory elements of the pattern $Q$ represent the concept $C$.

In this step, the rules that are concept equivalent will be grouped together. A group of rules which are all concept equivalent with respect to concept $C$ is said to produce the concept $C$.

The working memory element produced by the firing of a particular rule may represent more than one concept. Therefore, rules in a particular context can be divided in more than one possible way depending on the concepts of interest. Grouping of rules will be more complicated when the firing of each rule results in more than one working memory element and the concepts of interest are classified into a classification hierarchy (the groups may form a hierarchy). This step will be discussed in greater detail in the next section.

**Identify the patterns of working memory elements used and produced by each group**

A group $G$ of rules is said to produce (use) a working memory element of pattern $P$ if there exists a rule $R$ in $G$ such that $R$ produces (uses) a working memory element of the pattern $P$. If a group contains some other groups then the set of element patterns it produces (uses) is the result of the union of the sets of element patterns produced (used) by its enclosed groups.

The patterns of working memory elements produced by a particular group describe what changes could be made to the working memory when rules in the group are fired, while those used by the group are used to determine whether there is a rule with the satisfied condition part in the group. In this step, the element patterns produced and used by each group are identified. They will be used exclusively to analyze the interactions among groups in the next step.

**Analyze the relationships among the groups**

The purpose of this step is to identify the interactions among the groups of rules in the system. An affecting relation between two groups of rules is defined. Detecting this relation greatly helps the knowledge engineer to locate the effect of the firing of rules in a particular group on the whole system.
**Definition 14: Affecting Relation** Let $F$ and $G$ be groups of rules. $F$ affects $G$ if and only if there exist rules $R$ in $F$, and $S$ in $G$ such that the firing of $R$ can satisfy any of the following conditions.

1) Adds to the working memory a working memory element which matches a condition element of $S$.

2) Changes some attribute values of an existing working memory element $E$, which previously does not match any condition element of $S$, so that $E$ matches a condition element of $S$.

3) Changes some attribute values of an existing working memory element $E$, which previously matches a negated condition element $CE_n$ of $S$, so that $E$ does not match $CE_n$.

4) Removes from the working memory an existing working memory element which matches a negated condition element of $S$.

The patterns of working memory elements produced and used by a group summarize how the group influences other groups and how it is affected by other groups, respectively. Therefore, to determine whether group $F$ affects group $G$, it is sufficient to analyze the patterns produced by $F$ and used by $G$. In practice, $F$ affects $G$ if there exist patterns $P$ and $Q$ such that any of the following conditions is true.

1) $F$ produces $+P$, $G$ uses $Q$, and $P$ partially covers $Q$.

2) $F$ produces $P$, $G$ uses $Q$, and either $P$ partially covers $Q$ or $Q$ partially covers $P$.

3) $F$ produces $-P$, $G$ uses $-Q$, and either $P$ partially covers $Q$ or $Q$ partially covers $P$.

**Example**

The rules given in Figure 1 are taken from an OPS5 program for making credit decisions\(^5\) which consists of 40 rules. The problem-solving process of this program breaks down into five phases. A working memory element of class $\text{phase}$ is used exclusively to hold the current phase of the computation. According to the proposed methodology, it is first identified as a control element.

Since the rules over-twenty-good-economy, less-than-eighty-percent, less-than-five and greater-than-120-steady use the control elements of the same pattern, they are all context equivalent. Therefore, they are grouped together in one context. Similarly, the rules excellent downpayment and excellent-property are also context equivalent and are grouped together in another context.
(p over-twenty-good-economy
   (phase ^name categorization)
   (property ^money-down > <twp> ^twenty-percent <twp> ^region <r>)
   (economy ^price rising ^region <r>)
   {((loan ^downpayment nil) <loan>)
    → (modify <loan> ^downpayment excellent))
(p less-than-eighty-percent
   (phase ^name categorization)
   {((property ^price <eightyp> ^eighty <eightyp> ^appraised-value > 0) <p>)
    → (modify <p> ^home-value excellent))
(p less-than-five
   (phase ^name categorization)
   (property ^price <p> ^downpayment < <fp> ^five-percent <fp>)
   {((loan) <loan>)
    → (modify <loan> ^downpayment poor))
(p greater-than-120-steady
   (phase ^name categorization)
   {((property ^price < <otp> ^one-twenty <otp> ^home-value nil ^appraised-value > 0 ^region <r>) <p>
    (economy ^price falling ^region <r>)
    → (modify <p> ^home-value poor))
(p excellent-downpayment
   (phase ^name evaluation)
   {((loan ^downpayment excellent ^status undecided) <loan>)
    → (modify <loan> ^status accepted)
    (write (crif) [Accepted because of excellent downpayment] (crif)))
(p excellent-property
   (phase ^name evaluation)
   (property ^home-value excellent)
   {((loan ^status undecided) <loan>)
    → (modify <loan> ^status accepted)
    (write (crif) [Accepted because of excellent home value] (crif)))

FIGURE 1. OPS5 PRODUCTION RULES
The firing of rules over-twenty-good-economy and less-than-eighty-percent result in
the working memory elements, (loan ^{downpayment} excellent) and (property ^{home-value}
excellent), respectively. In the problem domain, these working memory elements can
be considered to represent the concept good-applicant. Therefore, these two rules are
grouped together. Similarly, the rules less-than-five and greater-than-120-steady are also
grouped together, since both the working memory elements (loan ^{downpayment} poor)
and (property ^{home-value} poor) can be considered to represent the concept not-good-
applicant.

Figure 2 depicts the contexts and groups of rules resulting from applying the proposed
methodology to the credit-decision program. Arrows in the figure denote affecting
relations among the groups.

![Figure 2. Groups of rules resulting from applying the methodology](image)

**CLASSIFICATION HIERARCHY OF CONCEPTS**

To systematically group the rules, the *strict classification hierarchy* of concepts, in which
any concept can be a specialization of only one concept, must be provided. In a strict
classification hierarchy of concepts, each primitive concept is defined by identifying the
patterns of working memory elements which represent it and other concepts are defined
as generalizations of primitive concepts.
[Definition 15: Primitive Concept] Concept $C_i$ in a classification hierarchy of concepts $H$ is called a primitive concept if and only if for any concept $C_j$ in $H$, $C_j$ is not a specialization of $C_i$.

[Definition 16: Direct Specialization] Let $C_i$ and $C_j$ be concepts in a classification hierarchy of concepts $H$. $C_i$ is a direct specialization of $C_j$ if and only if $C_i$ is a specialization of $C_j$ and for each concept $C_k$ in $H$, if $C_i$ is a specialization of $C_k$ then $C_k$ is not a specialization of $C_j$.

[Definition 17: Strict Classification Hierarchy] Let $H$ be a classification hierarchy of concepts. $H$ is called a strict classification hierarchy if and only if all the following conditions are satisfied.

1) For any primitive concepts $C_i$ and $C_j$ in $H$, if $C_i$ and $C_j$ are defined by element patterns $P$ and $Q$, respectively then $P$ and $Q$ are disjoint element patterns.
2) For each concept $C_k$ in $H$, $C_k$ is a direct specialization of at most one concept.

Given a particular set of working memory elements, several classification hierarchies of concepts may be constructed. Therefore, rules in one context could be grouped in more than one possible way depending on which hierarchy of concepts is of interest. Let $R, S$ be rules in the same context and $H$ be the strict classification hierarchy of concepts of interest provided. $R$ and $S$ will be grouped together if and only if for each element pattern $P_i$ produced by $R$ ($S$), if $P_i$ represents a primitive concept $C$ in $H$, then there exists an element pattern $P_j$ produced by $S$ ($R$) such that $P_j$ also represents $C$. An element pattern $P$ is said to represent a primitive concept $C$ if and only if $C$ is defined by an element pattern $Q$ and $P$ partially covers $Q$.

Some groups of rules should be further grouped together. Consequently, a rule base may be organized into a hierarchy of groups. Formally, two groups $F$ and $G$ of rules should be grouped together if all the following conditions are satisfied.

1) There exists a concept $C_i$ produced by $F$ and a concept $C_j$ produced by $G$ such that $C_i$ and $C_j$ are specializations of some concept $C_k$.
2) For each concept $C_i$ produced by $F$ ($G$), there exists a concept $C_m$ produced by $G$ ($F$) such that either $C_i$ and $C_m$ are the same concept or $C_i$ and $C_m$ are specializations of some concept $C_n$. 
ADVANTAGES OF THE PROPOSED METHODOLOGY

This section describes the advantages of applying the proposed methodology to the development of a production system in the aspects of understandability and maintainability improvement, rule-based verification and interference analysis.

Understandability and Maintainability Improvement

Let $R$ be a rule in group $G$. When the condition part of $R$ is modified, some element patterns used by $G$ may be changed. Consequently, some groups, which previously affected $G$, may not affect $G$ and some groups, which previously did not affect $G$, may now affect $G$. By observing the affecting relations changed, the effects of the modification on the whole system are located. On the other hand, when the action part of $R$ is modified, there are two possibilities, $R$ is still in $G$ or $R$ is now put into another group $F$. The effects of the modification on other groups in the first case can be determined by examining the changed affecting relations of $G$, while those of the second case by considering the altered affecting relations of $F$.

Consistency Verification

Inconsistencies in OPS5-based production systems are identified by detecting redundant rules, conflicting rules and subsumed rules. As a basis for presenting formal definitions of these relations between rules, the following terms are defined.

[Definition 18: Matching Set] A set of working memory elements, $M = \{E_1, E_2, \ldots, E_n\}$, is called a matching set of rule $R$ if and only if all the following conditions are satisfied.

1) There exists a bijective mapping from $M$ to the set of all non-negated condition elements of $R$, $N = \{CE_1, CE_2, \ldots, CE_n\}$, such that $E_i$ satisfies $CE_j$.
2) $E_1, E_2, \ldots, E_n$ simultaneously satisfy all non-negated condition elements of $R$.

[Definition 19: Equivalent Condition Parts] The condition parts of rules $R$ and $S$ are equivalent if and only if all the following conditions are satisfied.

1) For any set $M$ of working memory elements, $M$ is a matching set of $R$ if and only if $M$ is a matching set of $S$.
2) For any sets $M$ and $W$ of working memory elements, if $M$ is a matching set of $R(S)$ and $M$ is a subset of $W$ then the condition part of $R$ is satisfied if and only if the condition part of $S$ is also satisfied, when the working memory is $W$. 
[Definition 20: Positive Effecting Set] Let $R$ be a rule and $M$ be a matching set of $R$. A set $PE$ of working memory elements is called the positive effecting set of $R$ with respect to $M$ if and only if for each $E_i \in PE$, either $E_i$ is added by $R$ or $E_i$ is the result of changing some attribute values of $E_j \in M$ by $R$, when all non-negated condition elements of $R$ are matched by $M$.

[Definition 21: Negative Effecting Set] Let $R$ be a rule and $M$ be a matching set of $R$. A set $NE$ of working memory elements is called the negative effecting set of $R$ with respect to $M$ if and only if for each $E_i \in NE$, either $E_i$ is removed by $R$ or some attribute values of $E_i$ are changed by $R$, when all non-negated condition elements of $R$ are matched by $M$.

[Definition 22: Equivalent Action Parts] The action parts of rules $R$ and $S$ are equivalent if and only if for each matching set $M_i$ of $R$ ($S$), there exists a matching set $M_j$ of $S$ ($R$) such that the positive effecting set $PE_i$ and the negative effecting set $NE_i$ of $R$ ($S$) with respect to $M_i$ are equal to the positive effecting set $PE_j$ and the negative effecting set $NE_j$ of $S$ ($R$) with respect to $M_j$, respectively.

[Definition 23: Redundant Rules] Two rules are redundant if and only if they have equivalent condition parts and equivalent action parts.

[Definition 24: Conflicting Rules] Two rules are conflicting if and only if their condition parts are equivalent but their action parts are not equivalent.

[Definition 25: Subsumed Rules] Rule $R$ is subsumed by rule $S$ if and only if all the following conditions are satisfied.

1) For any matching set $M_i$ of $R$, there exists a matching set $M_j$ of $S$ such that $M_j$ is a subset of $M_i$.

2) For any matching sets $M_k$ of $R$, $M_l$ of $S$ and set $W$ of working memory elements such that $M_i$ is a subset of $M_k$ and $M_l$ is a subset of $W$, if the condition part of $R$ is satisfied then the condition part of $S$ is also satisfied, when the working memory is $W$.

3) Their condition parts are not equivalent but their action parts are equivalent.

In a resulting hierarchy of groups, it is obvious that two rules in different innermost groups are not redundant or subsumed rules, and rules in distinct contexts are not conflicting rules. Therefore, once a rule base is organized by applying the proposed methodology, the scope of detecting redundant rules and subsumed rules is within a single innermost group, and the scope of discovering conflicting rules is within a context.
Completeness Verification

A dead-end rule is a rule that does not activate the firing of any other rules and an impossible rule is a rule that cannot be activated by any other rules. The existence of a dead-end rule or an impossible rule indicates that there are missing rules and gaps in the knowledge base\textsuperscript{5,6,7} i.e. the knowledge base is not complete. In the present approach, the groups that do not affect any other groups and the groups that are not affected by any other groups are called dead-end groups and impossible groups, respectively. Dead-end groups and impossible groups can be removed from the rule base without any effect on the execution of rules in the rest of the system.

Parallel Rules Firing

There is interference among multiple rule firings when a parallel firing result is different from the result of any sequential firing. Based on the interference analysis using paired-rule conditions\textsuperscript{8}. The interference between two groups of rules can be detected. Let $F$ and $G$ be groups of rules. There is a possibility of interference between $F$ and $G$ if and only if there exist element patterns $P$ and $Q$ such that any of the following conditions is true.

1) $F (G)$ produces $+P$, $G (F)$ uses $-Q$, and $P$ partially covers $Q$.
2) $F (G)$ produces $P$, $G (F)$ uses $-Q$, and either $P$ partially covers $Q$ or $Q$ partially covers $P$.
3) $F (G)$ produces $-P$, $G (F)$ uses $Q$, and either $P$ partially covers $Q$ or $Q$ partially covers $P$.
4) $F (G)$ produces $+P$, $G (F)$ produces $-Q$, and $P$ partially covers $Q$.
5) $F (G)$ produces $P$, $G (F)$ produces $-Q$, and either $P$ partially covers $Q$ or $Q$ partially covers $P$.

SOPS PROGRAMMING ENVIRONMENT

Based on the proposed methodology, SOPS (Structured OPS5-based Production System), a new structured production-system programming language and environment has been designed and implemented. Basically two additions have been made to OPS5 to arrive at SOPS: one simple modification in the literalization section; and an addition of a new section, referred to as the concept classification section, for providing classification hierarchies of concepts.
To partition a rule base automatically into contexts, SOPS needs to identify certain working memory elements as control elements. The literalization section of OPS5 is modified so that the users can conveniently provide this information. The syntax of the literalization section of SOPS is formally defined in Figure 3. Apart from this, SOPS requires the strict classification hierarchy of concepts for automatically dividing rules in each context into groups. SOPS expects this hierarchy of concepts from the users through the newly introduced concept classification section. The syntax of the concept classification section is defined in Figure 4.

\[
\text{literalize-section} ::= \text{class-declaration}^* \\
\text{class-declaration} ::= (\text{literalize class-name attribute-name}* ) \\
\text{class-name} ::= \text{constant-symbolic-atom} \\
\text{attribute-name} ::= \text{constant-symbolic-atom}
\]

**FIGURE 3. SYNTAX OF LITERALIZATION SECTION OF SOPS**

\[
\text{hierarchy} ::= \text{concept}^* \\
\text{concept} ::= (\text{concept-name} = \text{pattern}^* ) \\
\text{pattern} ::= (\text{class-name attribute-specification}* ) \\
\text{attribute-specification} ::= ^\text{attribute-name attribute-restriction} \\
\text{attribute-restriction} ::= <\text{constant-symbolic-atom} >> \\
\text{atomic-restriction-value} ::= \text{constant-symbolic-atom} \\
\text{number} ::= \text{constant-symbolic-atom} \\
\text{generic-variable} ::= \text{constant-symbolic-atom} \\
\text{class-name} ::= \text{constant-symbolic-atom} \\
\text{attribute-name} ::= \text{constant-symbolic-atom}
\]

The terms \text{constant-symbolic-atom}, \text{generic-variable}, \text{number} and \text{predicate} have been left undefined. A \text{constant-symbolic-atom} is any sequence of characters, excluding certain special characters such as (,),{,},^; which does not have a form of a number or an operator. A \text{generic variable} is a variable of the forms *, **, *** etc. A \text{number} is either an integer number or a floating point number. A predicate is =,\text{>,<,<=,>=}, or ≥.

**FIGURE 4. SYNTAX OF CONCEPT CLASSIFICATION SECTION OF SOPS**
SOPS programming environment provides the features File, Literalization, Concept, Production, Analyze and OPS5 in its main menu. The feature File provides the user with file control functions. The features Literalization, Concept and Production offer built-in editors for the user to declare element classes, define classification hierarchies of concepts and enter production rules, respectively. The user can define more than one hierarchy of concepts and then specify the active hierarchy on which the current grouping of rules is based. The feature Analyze allows the user to list the groups affected, the groups activated, the groups interfered, the concepts produced, or the patterns of working memory elements used and produced by the selected group. Lastly, the OPS5 option invokes the translator which converts the existing SOPS program into its equivalent OPS5 program. An overview of SOPS programming environment is depicted in Figure 5.

![Diagram of SOPS Programming Environment]

**FIGURE 5. AN OVERVIEW OF SOPS PROGRAMMING ENVIRONMENT**

**RELATED RESEARCH WORK**

Jacob and Froscher proposed a methodology for organizing rules in OPS5-like production systems. Their methodology is also based on dividing rules into groups and focusing attention on the facts that carry information between rules in different groups. However, unlike SOPS, the basic approach they adopt for partitioning a rule base is to group together rules that affect each other and are likely to be changed at the same time. They defined the relatedness between two rules, which is measured by the number of non-control facts
that are mentioned in both rules, and proposed a clustering algorithm that automatically groups related rules together. In contrast to their work, SOPS automatically groups the rules based on a hierarchy of concepts provided by a user. That is, rules are grouped together if they produce working memory elements the patterns of which are specified to represent same concepts in a given classification hierarchy. By providing different hierarchies of concepts, a SOPS user can partition a particular rule base in various ways depending on the hierarchy of concepts in which he/she is currently interested. This gives a higher level of abstraction in studying different flows of concepts in the knowledge base.

Agarwal and Tanniru\textsuperscript{9} presented another approach for managing the complexity of a production rule base. In their approach, rules are first synthesized according to their syntax into a directed graph called a parameter dependency network. The resulting network is then simplified, by aggregating parameters into groups of parameters, at successive levels of abstractions in order to understand the major relationships between input parameters and goal parameters at different levels of details. However, as remarked in their conclusions, the specification of the parameter dependency network described in reference 9 does not provide parameter details at the level of values. Information about the feasible range of values of a parameter cannot be explicitly represented. Therefore, the network does not allow reachability analysis and identification of missing rules. In SOPS, rules are first grouped according to their semantics, i.e. rules which produce different working memory elements but the same concept are also grouped together, and dependencies among the resulting groups are then analyzed. A concept is identified by patterns, which can be values, of working memory elements. This allows reachability analysis of concepts as well as groups of rules.

Some research groups have attacked the problem of consistency and completeness verification in rule-based systems. Nguyen et al.\textsuperscript{6} implemented a program, called CHECK to verify the consistency and completeness of knowledge bases built for the Lockheed expert system (LES) shell. However, the definitions and techniques used in CHECK are found to be unsuitable for verification in OPS5-like production systems because the actions in OPS5 are non-monotonic and the condition elements in OPS5 are both existentially (in positive clauses) and universally (in negated clauses) quantified\textsuperscript{7}.

Prakash et al.\textsuperscript{7} have described a methodology for verification of OPS5-based AI applications, which is based on converting the condition and action parts of rules into a linear system of inequalities and equalities and testing them for feasible solution. In their work, two relations between rules, called conflict and likely-to-activate
relations, and two properties of rules, called dead-end and impossible rules, are defined. They claimed that discovering these relations and properties by compile-time analysis is useful for detecting errors in production systems. In SOPS, the affecting relations are defined similar to their likely-to-activate relations, but between two groups of rules. By analyzing affecting relations among groups of rules at compile time, the presence of dead-end and impossible groups can be detected. That is, their work considers rules as basic units while SOPS considers groups of rules as basic units in analyzing a rule base.

Ishida\textsuperscript{9} provided an algorithm resulting from combining compile-time and run-time techniques for detecting interference among multiple rule firings. The algorithm is based on analyzing a data dependency graph of rules in a production system. The compile-time analysis of interference among groups of rules implemented in SOPS permits less parallelism than his algorithm. However, if two groups of rules were determined not to cause interference by SOPS, a precise check of interference between rules in these groups at run-time is not necessary.

CONCLUSIONS

A new methodology for developing structured production system is proposed. In this methodology, rules are divided into groups based on their contexts and the concepts they produce. The patterns of working memory elements that influence the firing of rules and those that are produced by rules in a group are then identified. By analyzing these element patterns, the interactions among groups of rules are easily determined. The hierarchy of groups resulting from applying this methodology allows the verification of correctness, consistency and completeness of the rules in the knowledge base as well as subsequent knowledge base maintenance. Based on the proposed methodology, a new programming language and environment referred to as SOPS has been developed and implemented. SOPS facilitates viewing the production rules in various ways by allowing the user to specify different classification hierarchies of concepts.

Two further works should be investigated. First, a number of experiments which apply the methodology to real production systems consisting of a large number of rules should be performed. Second, to make this work more theoretically sound, theories about various properties of rules in the groups resulting from the application of the methodology should be formally developed.
ACKNOWLEDGEMENT

The authors gratefully acknowledge the partial financial support provided by the National Electronics and Computer Technology Centre (NECTEC), the National Science and Technology Development Agency of Thailand, to carry out this research work.

REFERENCES


8 Ishida, T. Parallel Rule Firing in Production Systems. IEEE Transactions on Knowledge and Data Engineering, 1991, 3(1).