A Methodology for Systematic Verification of OPS5-Based AI Applications

G.Ravi Prakash[1], E.Subrahmanian[2], and H.N.Mahabala[1]

[1]Dept. of Computer Science and Engineering
Indian Institute of Technology, Madras - 600036, India
e-mail : ravi@shiva.emet.in

[2]Engineering Design Research Centre
Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail : sub@edrc.cmu.edu

Abstract

One of the critical problems in putting AI applications into use in the real world is the lack of sufficient formal theories and practical took that aid the process of reliability assessment. Adhoc testing, which is widely used as a means of verification, serves limited purpose. A need for systematic verification by compile-time analysis exists. In this article, we focus our attention on OPS5-based AI applications and present a methodology for verification which is based on compile-time analysis. The methodology is based on the principle of converting the antecedent and action-parts of productions into a linear system of inequalities and equalities and testing them for a feasible solution. The implemented system, called SVEPOA, supports interactive and incremental analysis.

1 Introduction

Reliability assessment of artificial intelligence (AI) applications is an important problem. One of the bottlenecks in taking AI applications to the end-user sites is the lack of sufficient formal theories and practical tools that aid the process of reliability assessment. AI applications attempt to automate (to higher degrees) intelligent decision making activities of humans. The consequences of errors in AI applications are likely to be more serious or costlier than those in conventional computer applications. Adhoc testing, which is widely used as means of verification, serves limited purpose. It, however large in volume, can only reveal the presence of errors but does not ensure their absence. A need for systematic verification of AI applications exists.

We present in this paper a methodology for systematic verification of OPS5-based AI applications. We have focused our attention on production systems [Newell72] in general and OPS5-like languages [Forgy81 in particular for reasons that production system model of reasoning is one of the earliest and widely used models by the AI researchers and many interesting prototype applications [e.g. McDermott82] have been developed using OPS5-like languages. We present here a methodology that is based on compile-time analysis. Through compile-time analysis, we discover certain properties and relations of productions and illustrate through discussions that the presence and absence of these properties and relations reveal errors, if any, in the design of the antecedent and action-parts of productions.

2 Production System Model and OPS5 Language

A Production System, Z, can be characterized by three components : Z = (D, P, C) where,

D is a Fact-base : a set of facts;

P is a Production-base : a set of productions;

C is a Control strategy : a set of search procedures.

Production systems can be classified based on

* the scheme of representation used for the objects, attributes, relationships, and states in the fact-base;

* the nature of the actions in the productions, i.e. monotonic (ignorable) or nonmonotonic (revocable or irrevocable) actions;

* the techniques used for control strategy (e.g. weak search methods or heuristic search methods).

Based on this classification, the OPS5 language and its underlying production system model can be characterized as :

1. OPS5 represents fact-base as a set of instantiated objects having values to (some of) their attributes;

2. antecedent-parts of productions in OPS5 are expressed as a conjunction of clauses *(positive* and *negated* clauses) where each clause is represented as a conjunction of one or more ordinal predicates involving attributes and constants/variables;

3. actions in productions of OPS5 are nonmonotonic; the actions are mainly two types : *make* to add new instantiations of objects to the fact-base and *remove* to delete existing ones *(modify* action can be treated as a combination of *make* and *remove),*

4. OPS5 offers MEA and LEX control strategies and *data-driven* (forward) reasoning.

This characterization of OPS5 is used in course of discussions in the rest of this paper.

3 Related Research Work

The research work reported earlier in [Nguyen85, Mahabala87, Ginsberg88 etc.] on the issue of consistency checking of production (or rule-based) systems are all related to monotonic production systems that use propositional formulae to represent fact-base and productions. The definitions and the techniques used in the verification by the above referenced work are found to be unsuitable for verification of OPS5-like production systems for two principal reasons : one is that the actions in OPS5 are nonmonotonic and the other is that the antecedent-parts in OPS5 productions can use predicates involving existential (in positive clauses) and universal (in negated clauses) quantifiers. Verification of OPS5-based applications by compile-time analysis requires entirely new techniques.

4 Sources of Errors in the Design of Productions

The process of designing an AI application using production system model involves translating domain knowledge into fact-base declarations and productions. During this process, often (excluding toy problems) the knowledge-engineer makes assumptions about what knowledge is to be stated explicitly (called qualification and ramification problems). As the fact-base and production-base are enlarged and refined, the knowledge-engineer needs to maintain the consistency of assumptions. Otherwise the production system is prone to contain errors. This is one source of errors. There are other sources of errors too, like the text-editing mistakes, the knowledge being often incomplete, etc.

5 Verification by Compile-time Analysis

We have manually analyzed toy and prototype AI application systems developed using OPS5. We have found that discovering the following relations and properties of productions by compile-time analysis helps the knowledge engineer to detect errors in the antecedent and action-parts of productions :

1] conflict relations
2] likely-to-activate relations
3 dead-end productions
4] impossible productions

In this section, we formally define these relations and properties and discuss their usefulness in the process of verification. Before we do so, we introduce certain terms which we use in defining the above listed relations and properties.

Activating and Resultant Sets :

The set of all valid fact-base states of a given OPS5-based AI application is called its state-space. Each production in the application system can be viewed as a state transformational operator (the term 'state' implies 'valid fact-base state'). Every production specifies a set of conditions (antecedent-part), which when satisfied, makes the production eligible for execution. The number of states that can satisfy the antecedent-part of a production is *zero* or *more* depending upon the logical and semantic consistency and specificity of conditions in the antecedent-part. We introduce here two terms, Activating-Set and Resultant-Set of a production.

Activating-Set (A_p) of a production, p :

$$A_p \equiv \{ s_i \mid s_i \text{ is a state in state-space AND } s_i \text{ satisfies the antecedent-part of the production, p } \}$$

Activating-Set of a production, p, is the set of all states in the state-space that satisfy the antecedent-part of p.

Resultant-Set (R_p) of a production, p :

$$R_p \equiv \{ s_j \mid s_j \text{ is a state in state-space AND } s_j \text{ is obtained by applying the actions of the production, p, on some state } s_i \in A_p \}$$

Resultant-Set of a production, p, is the set of all states in state-space, each of which results from applying the actions of production p onto some state s, belonging to the Activating-Set of p.

Using the definitions for Activating-Set and Resultant-Set of a production, the relations and properties of a production as listed earlier in this section are defined below.

Conflict Relation :

A conflict relation is said to exist between two distinct productions p and q iff,

$$((A_p \cap A_q) \neq \phi)$$

Discussion : Conflict relation exists between two distinct productions, p and q, (denoted as 'p.conflict.q') iff, the Activating-Sets A_p and A_q have common elements. In other words, there exist one or more fact-base states that can satisfy the antecedent-parts of both p and q. When solving AI problems using OPS5, the conflict-set (the set of productions that are satisfied by the current fact-base) often contains more than one production. So conflict relations between productions is a natural phenomenon in OPS5-based applications. But, while designing productions, the knowledge engineer may *over* or *under* specify the conditions in the antecedent-parts (because of qualification problem or incomplete knowledge). This gives rise to the possibility that the OPS5-based application system contains certain invalid conflict relations or does not contain certain valid conflict relations. To illustrate this point, consider the two productions given in example.1 (these productions are selected from the solution to the monkey and banana problem given in Appendix-One of [Brownston85] for easy readability). Consider that the contents of the fact-base are described by the test data given in example.2.

Example.1

```
(p  Holds::Object-NotCeil:On
    (goal ^status active ^type holds
                         ^object-name <o1>)
    (phys-object ^name <o1> ^weight light
                 ^at <p> ^on <> ceiling)
    (monkey ^at <p> ^on <> floor)
-->
```

```
   (make goal ^status active ^type on
              ^object-name floor) )
(p  Holds::Object:Holds
    (goal ^status active ^type holds
                         ^object-name <o1>)
    (phys-object ^name <o1> ^weight light
                            ^at <p>)
    (monkey ^at <p> ^holds {<>nil <> <o1>})
-->
    (make  goal ^status active ^type holds
                ^object-name nil) )
```

Example.2

```
(make phys-object ^name banana ^weight light
                  ^on ceiling ^at 3-3)
(make phys-object ^name ladder ^weight light
                  ^on floor ^at 1-1)
(make monkey ^on ladder ^holds blanket
                  ^at 1-1)
(make phys-object ^name blanket ^weight light
                  ^at 1-1)
(make goal ^status active ^type holds
                  ^object-name ladder)
```

The fact-base describes that there exists a goal for the monkey to hold the ladder; the ladder is on the floor at position 1-1; the monkey is at position 1-1, on the ladder and holding the blanket. Both the productions of example.1 are satisfied by the fact-base of example.2. So, a conflict relation exists between these two productions. Though the authors of Appendix-One of [Brownston85] very clearly state that there are no conflicting productions to either of these productions, we could find a conflict relation between them using our systematic compile-time analysis. Similarly, one can find that the production 'Holds::Object:Satisfied' has a conflict relation with the production 'Holds::Object-NotCeil:On' (both productions from Appendix-One of [Brownston85]), which the authors stated as impossible. What conflict relations are valid, and what are not, is a domain dependent feature. It is not possible to determine invalid conflict relations domain independently for (nonmonotonic) production systems built using OPS5-like languages (this is in contrast to monotonic production systems using propositional formulae, in which it is possible to define and identify conflicting productions domain-independently ('if p then q' and 'if p then NOT q') through syntactic analysis). Besides the presence of invalid conflict relations, the absence of valid conflict relations is also a cause of major concern to the knowledge engineer debugging a production system. We define a production system to be in error if it contains invalid conflict relations or does not contain valid conflict relations. Our contention is that if there are errors in a toy system (solution to the monkey and banana problem) of around 24 productions, the likelihood of errors in any real-world system of 2500 productions or more is much higher. Hence, a systematic compile-time analysis is much in place.

Likely-to-activate Relation :

A likely-to-activate relation is said to exist from a production, p to a production, q (q not necessarily different from p), iff,

$$((R_p \cap A_q) \not\equiv \phi)$$

Discussion : Likely-to-activate relation exists from a production, p to a production, q iff, the Resultant-Set of p, Rp, and the Activating-Set of q, Aq, have common elements. In other words, the execution of p can result in a fact-base state, such that, that fact-base state satisfies the antecedent-part of q. This relation is denoted by 'p.lta.q' (lta as an acronym for likely-to-activate). The process of finding a solution to a problem using production system model involves finding a path (the ordered set of likely-to-activate relations) from the initial fact-base state to the desired goal fact-base state. One of the major concerns of a knowledge engineer during the process of debugging is to find the invalid paths (or invalid likely-to- activate relations) taken by the system. Since adhoc testing cannot reveal the presence of all the invalid likely-to-activate relations, a systematic compile-time analysis is required. As with the conflict relations, what likely-to-activate relations are valid and what are not is a domain dependent feature. Besides the presence of invalid likely-to-activate relations, the absence of valid likely-to-activate relations also reflect errors. We define a production system to be in error if it contains invalid likely-to-activate relations or does not contain valid likely-to-activate relations.

Dead-end Productions :

A production, p, is said to be a dead-end production, iff

(1) ∀ (q ∈ production-base) (p.lta.q is false) AND
(2) the action-part of p does not include an 'halt' action;

Discussion : A production, p is said to be a dead-end production iff, p does not have an lta relation to any other production, q, in the production-base and the action-part of p does not include an 'halt' action. While testing OPS5 applications, the knowledge engineer sometimes comes across a situation in which the execution of a production causes the conflict-set to be empty in the next cycle. Such situations are termed as dead-end situations and the productions that cause such situations are called dead-end productions. Since adhoc testing may not reveal all the dead-end situations, a compile-time analysis is needed.

The above definition of dead-end production is a weak definition. Productions satisfying the above definition are definitely dead-end productions. A production, p, not satisfying that definition can also be dead-end productions, if there are one or more states in Rp not belonging to A_q, for any other production, q.

We can define a production to be a pseudo-dead end production if its action-part includes an 'halt' action and it has atleast one lta relation to another production.

Impossible Productions :

A production, p, is said to be an impossible production, iff

(1) $(A_p \equiv \phi)$ OR
(2) $(A_p \cap S_{ini} \equiv \phi \wedge \neg\exists\, q\ (q.lta.p))$

where S_{ini} is the set of all possible initial states.

Discussion : A production, p, is said to be an impossible production, iflf 'the Activating-Set of p is empty' or 'p is not a start production and no other production in the production-base has an Ita relation with p'. The Activating-Set of any production will be empty when the set of conditions in the antecedent-part of p are either logically or semantically inconsistent. Impossible productions never get instantiated into the conflict-set and are difficult to be detected by an adhoc testing method.

6 Computational Feasibility

The computational feasibility of finding the relations and the properties (described in Section 5) of a production depends on the nature of predicates that can be used in the antecedent and action-parts of productions. From a study of the BNF description of the syntax of OPS5 language given in [Brownston 85], we have the following information. Predicates that can be used in the antecedent-part are of two types : (X^i.R.k) or (X_i.R.X_j) where X_i, X_j are attributes of objects declared by 'literalise' commands; R is one of the ordinal relations; R G { = ,<> , < , < , > , > } ; k is a constant (string, integer or real depending on X_i).

Predicates that can be used in the action-part are of three types : (X_i = k) , (X_i = Xy) , or (X_i = exp) where X,, Xy, and k are the same as above and exp is any arithmetic expression.

Assumptions :

We make an assumption that the arithmetic expressions that can appear in the predicates of action-part are linear arithmetic expressions (that is, they do not involve non-linear expressions like (X_i,*X_i) or (X_i,*X_j*X_k*) etc.). This assumption is fairly reasonable for two reasons : [i] many AI applications involve mostly symbolic reasoning rather than evaluating non-linear arithmetic expressions; [ii] through this assumption, the divide and conquer principle is used and the systems that involve predicates with only linear expressions are given an effective procedure for compile-time analysis. We also assume that for each attribute of every object declared by the literalise command, the range of values the attribute can take are finite. This assumption is realistic for many real world problems. We also like to note here that

* boolean constants, "true" and "false" can be encoded as integers 1 and 0 respectively;

* symbolic constants can be sorted in lexicographic order and mapped onto the natural numbers; and

* boolean/symbolic variables can be treated as integer variables.

Based on the above assumptions, the problem of finding the relations and the properties of productions reduces to the problem of finding whether there exists a feasible solution to a system of linear inequalities and equalities with integer variables. This is a known and solvable problem. We use the algorithm given in [Biswas87] that exploits the simplicity of the predicates. [Biswas87] offers a polynomial-time algorithm to find the feasibility if all the inequalities and equalities are 'simple' (please refer to [Biswas87] for the definition of 'simple' inequalities and equalities). We have observed in analysing protype applications that most often the antecedent and action-parts of productions contain 'simple' inequalities and equalities.

7 Implementation of SVEPOA

We have implemented our methodology in common LISP on MicroVaxII workstation. This program, called SVEPOA (stands for Systematic VErification Program for 0PS5-based Applications) has the following two main features :

1] Interactive Analysis Facility
[2] Incremental Analysis Facility

Interactive Analysis Facility : We observe that knowledge engineers often try to structure and segment an OPS5-based application and hence it may not be required to compare roductions in one segment with productions in other segments exhaustively. The Interactive Facility offers the knowledge engineer a method of choosing which properties (or relations) to discover in (or across) what segments of productions.

Incremental Analysis Facility : When SVEPOA is initially run, it stores the set of relations and properties discovered. In the subsequent analysis, SVEPOA uses and updates the set of relations and properties pertaining to an application.

The main subprograms of SVEPOA are find-conflict-relation, find-Ita-relation, find-if-dead-end, find-if-impossible and find-feasibility. The functions of SVEPOA and its subprograms are described in the following procedures. A trace of the procedures for an example is given in Appendix-A.

procedure SVEPOA;

1. Ascertain the objects, their attributes and the range(s) of values for each attribute;
2. Sort the symbolic constants in lexicographic order and map them onto the natural numbers;
3. Ascertain the production(s) and the property or relation to be discovered; call the appropriate subprocedure;
4. Repeat step 3 until no more analysis is required;

subprocedure find-conflict-relation(p,q);

1. Find Cc, the conjunction of the antecedent-parts of p (Cp) and q (Cq); i.e Cc = (Cp) ∧ (Cq);
2. Call subprocedure find-feasibility (Cc) to find out if there exists a feasible solution to Cc;
3. If Cc has a feasible solution then (p.conflict.q) is true else not;

subprocedure find-Ita-relation(p,q);

Let Cp, Dp and Cq be the antecedent-part of p, action-part of p and antecedent-part of q respectively;

1. Convert each *modify* action in Dp into an equivalent combination of *remove* and *make* actions;
2. For each *remove* action, r, in Dp, delete the clause referenced by r from Cp;
3. For each *make* action, m, in Dp, add a clause containing the '=' predicates of m to Cp;
4. Cc = (Cp) ∧ (Cq);

5. Call subprocedure find-feasibility(Cc);

6. If Cc has a feasible solution then (p.lta.q) is true eke not;

subprocedure find-if-dead-end(p);

1. Find if the action-part of p (Dp) contains an 'halt' action;

2. If 'halt' action is not included in Dp then find out if there is atleast one other production, q, such that (p.lta.q) is true;

3. If 'halt' action is included in Dp then find out whether p is a pseudo-dead-end production; this is done by finding out if there is atleast one other production, q, such that (p.lta.q) is true;

subprocedure find-if-impossible(p);

1. Find out whether ($A_p \equiv \phi$) or not by testing if the antecedent-part of p (Cp) has a feasible solution;

2. If ($A \not\equiv \phi$) then find out whether (($A_p \cap S_{ini}$-) $\equiv \phi$) or not; S_{ini} is expressed as a conjunction of input predicates describing the possible initial values of the attributes of the objects; the attributes not appearing in S_{tni}) can take any value from their ranges;

3. If (($A_p \cap S_{tni}$) $\equiv \phi$) then find out if there is atleast one other production such that (q.lta.p) is true;

subprocedure find-feasibility (Cc);

1. Conver the conditions in the clauses of Cc into the form (object,attribute R constant) or (object,attribute R variable) where R is the ordinal relation;

2. Each distinct object.attribute is treated as a variable while using Biswas algorithm; All symbolic constants are replaced by the corresponding natural numbers;

3. Apply DeMorgan's law onto each negated clause in Cc; for example, a negated clause of the form

-((phys-object.on = <o1>)∧(phys-object.weight = heavy))

is converted to :

((phys-object.on <> <o1>) ∨ (phys-object.weight <> heavy))

4. Each '<>' condition in Cc is expressed as a disjunction of '<' , '>' relations;

5. Cc is converted into an equivalent disjunctive normal form expression; each disjunct, t, and the range of values of the unresolved variables (variables equated to constants are resolved variables) are passed onto the Biswas algorithm to find if t has a feasible solution;

8 Conclusions

Since adhoc testing cannot reveal all errors, a compile-time analysis to systematically verify AI applications is very much needed. We presented here a methodology that analyzes OPS5-based applications at compile-time and detects certain relations and properties of productions. Discovering the presence and absence of these relations and properties helps the knowledge engineer to find errors in the antecedent and action-parts of productions. Our methodology and the tool have the advantages as well as the disadvantages of a domain-independent analysis. Being domain-independent, our tool does not require formal specifications of the application. Asking for formal specifications of AI applications is often not only difficult but also limits the applicability of a tool to some specific classes of problems. Our tool, being domain-independent, can be used for analyzing any OPS5-based AI application. However, it has the limitation of being able to only point out *possible* errors but not *definite* errors. Abo, we don't claim that the analysis done by our tool is complete. There may be other relations and properties of productions, detecting which will be useful during verification.

Acknowledgments : We express our sincere thanks to Richard Christie, Petter Stoa, Prof.S.N.Talukdar and Prof.C.R.Muthukrishnan for the useful discussions with them.

Appendix-A

We show the trace of the important steps of SVEPOA to find the conflict relation between productions of Example. 1 of Section 5.

procedure : SVEPOA

1. Objects are goal, phys-object and monkey; these objects are denoted by O1, O2 and O3 respectively; the range(s) of values of the attributes of these objects are obtained; for example,

range(O2.name) = [bananas, blanket, couch, ladder];

range(O3.at) = [1-1, 1-2, . . . 10-10];

2. All the symbolic constants are sorted and mapped onto the natural numbers; for example, 1-1 → 1; 1-2 → 2; 10-10 → 100; active → 101; etc.

3. Let the two productions 'Holds::Object:Not Ceil:On' (ref. to as p) and 'Holds::Object:Holds' (ref. to as q) are accepted;

find-conflict-relation(p,q);

subprocedure : find-conflict-relation(p,q);

1. Cc = (Cp) ∧ (Cq);

find-feasibility(Cc);

subprocedure : find-feasibility(Cc);

1. The conditions in Cc are expressed in the forms (object.attribute R constant) or (object.attribute R variable); for example, the first clause in Cc is expressed as

((goal.status = active) ∧ (goal.type = holds) ∧ (goal. object-name = <o1>));

2. Each distinct object.attribute in Cc is treated is a variable; let the variables be denoted as Z11 (for goal.status), Z12 (for goal.type), Z13, etc. in the order respectively; all symbolic constants are replaced by their corresponding natural numbers; for example (monkey.on <> floor) will now appear in Cc as (Z32 <> 107) as "floor" is mapped onto 107 in this domain; now Cc is :

Cc ≡ ((Z11 = 101) ∧ (Z12 = 109) ∧ (Z13 = <01>) ∧ (Z21 = <01>) ∧ (Z22 = 111) ∧ (Z23 = <p>) ∧ (Z24 <> 105) ∧ (Z31 = <p>) ∧ (Z32 <> 107) ∧ (Z33 <> 113) ∧ (Z33 <> <01>));

3. There are no negated clauses in Cc;

4. Each '<>' condition in Cc is expressed as a disjunction of '<' , '>' relations; for example, (Z32 <> 107) is expressed as ((Z32 < 107) ∨ (Z32 > 107));

5. Cc is converted into an equivalent disjunctive normal form expression; each disjunct (there are 16 of them in this case) along with the range of values of the unresolved variables is passed onto the Biswas algorithm; one of the disjuncts of the dnf expression is :

t ≡ ((Z11 = 101) ∧ (Z12 = 109) ∧ (Z13 = <01>) ∧ (Z21 = <01>) ∧ (Z22 = 111) ∧ (Z23 = <p>) ∧ (Z24 > 105) ∧ (Z31 = <p>) ∧ (Z32 > 107) ∧ (Z33 < 113) ∧ (Z33 < <01>));

The unresolved variables are Z13, Z21, Z23, Z24, Z31, Z32, Z33; all these inequalities are 'simple' and Biswas algorithm returns the following solution :

Z13 = 110; Z21 = 110; Z23 = 1-1; Z24 = 107; Z31 = 1-1; Z32 = 110; Z33 = 104;

Interpretation :

goal.status = active;
goal.type = holds;
goal.object-name = ladder;
phys-object.name = ladder;
phys-object.weight = light;
phys-object.at = 1-1;
phys-object.on = floor;
monkey.at = 1-1; monkey.on = ladder;
monkey.holds = blanket;
So, (p.conflict.q) is true;

References :

[Biswas87] : Biswas,S. and Rajaraman,V. , 'An Algorithm to Decide Feasibility of Linear Integer Constraints Occurring in Decision Tables', IEEE Tran, on Software Engg., vol SE-13:1340-1347, Dec,1987

[Brownston85] : Brownston,L., Farrell,R., Kant,E., Martin,N., 'Programming Expert Systems in OPS5', Addison-Wesley Pub., 1985

[Forgy81] : Forgy,C.L., 'OPS5 User's Manual', Carnegie-Mellon Dniv., CMU-CS-81-135, July 1981

[Ginsberg88] : Ginsberg,A., 'Knowledge-Base Reduction : A new Approach to Checking Knowledge Bases for Inconsistency Redundancy', Proc. of AAAI Conf.,:585-589, August, 1988, St Paul, Minnasota, USA

[Mahabala87] : Mahabala,H.N., Ravi Prakash,G., et al, 'Expert System for Selection of DriII: A Case Study for use of Metaknowledge and Consistency Checks', Proc. of the II Int. conf. on Appln. of AI in Engg.', Boston, pp 371-386, 1987

[McDermott82] : McDermott,J., 'RI : A Rule-based Configurer of Computer Systems', AI Journal, vol.19, Sept., 1982

[Newell72]: Newell,A. and Simon,H.A., 'Human Problem Solving', Prentice-Hall Inc., 1972

[Nguyen85] : Nguyen, T. et. al, 'Checking Expert System Knowledge Base for Consistency and Completeness', Proc. of IJCAI-85 : 375-378, 1985