

CHECKING AN EXPERT SYSTEMS KNOWLEDGE BASE FOR CONSISTENCY AND COMPLETENESS

T. A. Nguyen, W. A. Perkins, T. J. Laffey,
and D. Peora

Lookheed Research and Development
0/92-10 B/254E
3251 Hanover Street, Palo Alto, CA. 94304

ABSTRACT

In this paper we describe a program that verifies the consistency and completeness of expert system knowledge bases which utilize the Lookheed Expert System (LES) framework. The algorithms described here are not specific to LES and can be applied to most rule-based systems. The program, called CHECK, combines logical principles as well as specific information about the knowledge representation formalism of LES. The program checks for redundant rules, conflicting rules, subsumed rules, missing rules, circular rules, unreachable clauses, and deadend clauses. It also generates a dependency chart which shows the dependencies among the rules and between the rules and the goals. CHECK can help the knowledge engineer to detect many programming errors even before the knowledge base testing phase. It also helps detect gaps in the knowledge base which the knowledge engineer and the expert might have overlooked. A wide variety of knowledge bases have been analyzed using CHECK.

1.0 INTRODUCTION

The Lookheed Expert System (LES) is a generic rule-based expert system tool [1] (similar to EMYCIN [2]) that has been used as a framework to construct expert systems in many areas such as electronic equipment diagnosis, design checking, photo interpretation, and hazard analysis. LES employs a combination of goal-driven and data-driven rules with the latter being attached to the factual database (demons). One objective in the design of LES was to make it easy to use. Thus, many debugging tools and aids were added to the LES program. One of these aids is the knowledge base completeness and consistency verification program called CHECK. Its purpose is to help a knowledge engineer check the knowledge base which he constructed for logically redundant rules, conflicting rules, subsumed rules, missing rules, unreachable clauses, and deadend clauses. CHECK does not perform any syntax checking on the rules, since this is done automatically when the rule files are loaded into the knowledge base. CHECK

statioally analyzes the knowledge base (i.e., after the rules and facts are loaded into the knowledge base), unlike TEIRESIAS, which performs an assessment of rules in the setting of a problem-solving session [3].

Suwa, Soott, and Shortliffe [4] have written a program for verifying that a set of rules comprehensively spans the knowledge base. This program was devised and tested within the context of the ONCOCIN system (an EMYCIN-like system). Our work differs from theirs in that CHECK includes unreachable clauses and deadend clauses as two additional rule checking criteria. Furthermore, CHECK produces a dependency chart and detects any circular rule chains. Also, CHECK was devised and tested on a generic expert system with case-grammar rules and a "frame" database. It has been used to analyze a wide variety of knowledge bases. CHECK does not take into account certainty factors when checking the rule base.

2.0 CHECKING FOR CONSISTENCY AND COMPLETENESS

A static analysis of the rules can detect many potential problems and gaps that exist in a rule base. We now identify and give definitions for seven criteria that are used by CHECK to perform static analysis of any rule base constructed for use with LES. The first four criteria are concerned with potential problems, whereas the last three criteria are concerned with gaps in the knowledge base.

2.1 POTENTIAL PROBLEMS IN A KNOWLEDGE BASE

By statically analyzing the logical semantics of the rules represented in LES's case grammar format, CHECK can detect redundant rules, conflicting rules, rules that are subsumed by other rules, and circular-rule chains. The following definitions for these four potential problems are used in CHECK:

- o Redundant rules: two rules succeed in the same situation and have the same results. In LES, this means that the IF parts of the two rules are

equivalent, and one or more THEN clauses are also equivalent. Because LES allows variables in rules, equivalent means that the same specific object names can match their corresponding variables. For example the rule " $p(x) \rightarrow q(x)$ " is equivalent to the rule " $p(y) \rightarrow q(y)$ ", where x and y are variables.

- o Conflicting rules: two rules succeed in the same situation but with conflicting results. In LES, this means that the IF parts of the two rules are equivalent, but one or more THEN clauses are contradictory, or one pair of IF clauses is contradictory while they have equivalent THEN clauses. For example, the rule " $p(x) \rightarrow \text{not}(q(x))$ " is contradictory to the rule " $p(x) \rightarrow q(x)$ ".
- o Subsumed rules: two rules have the same results, but one contains additional constraints on the situations in which it will succeed. In LES, this means one or more THEN clauses are equivalent, but the IF part of one rule contains fewer constraints and/or clauses than the IF part of the other rule. For example, the rule " $(p(x) \text{ and } q(y)) \rightarrow r(z)$ " is subsumed by the rule " $p(x) \rightarrow r(z)$ ".
- o Circular rules: a set of rules is a circular-rule set if the chaining of those rules in the set forms a cycle. For example, if we had a set of rules as follows: (1) " $p(x) \rightarrow q(x)$ " (2) " $q(x) \rightarrow r(x)$ " (3) " $r(x) \rightarrow p(x)$ " and the goal is $r(A)$, where A is a constant, then the system will enter an infinite loop at run time, unless the system has a special way of handling circular rules.

2.2 POTENTIAL GAPS IN A KNOWLEDGE BASE

The development of a knowledge-based system is an iterative process in which knowledge is encoded, tested, added, changed, and refined. This iterative process often leaves gaps in the knowledge base which both the knowledge engineer and the expert may have overlooked during the knowledge acquisition process. In LES, we have found three situations indicative of gaps in the knowledge base. These three situations, called CD missing rules, (2) unreachable clauses, and (3) deadend clauses are described below:

- o Missing rules: a situation in which some values in the set of possible values (called legal values) of an object's attribute are not covered by any rule's IF clauses (i.e., the legal values in the set are covered only partially or not at all). A partially covered attribute can prohibit the

system from attaining a conclusion or cause it to make a wrong conclusion when an uncovered attribute value is encountered during run time.

- o Unreachable clauses: in a goal-driven production system, a THEN clause of a rule should either match a goal clause or match an IF clause of another rule (in the same rule set). Otherwise, the THEN clause is unreachable.
- o Deadend Clauses: to achieve a goal (or subgoal) in LES, it is required that either: (1) the attributes of the goal clause are askable (user provides needed information) or (2) that the goal clause is matched by a THEN clause of one of the rules in the rule sets applying to that goal. If neither of these conditions is satisfied then the goal clause can not be achieved, i.e., it is a "deadend clause". Similarly, the IF clauses of a rule also must meet one of these two conditions, or they are "deadend clauses".

2.3 DEPENDENCY CHART AND CIRCULAR-RULE CHAINS DETECTION

As a by-product of the rule checking, CHECK generates a dependency chart which shows the interactions among the rules and between the rules and the goal clauses. An example of a dependency chart for a small problem is shown in Figure 1. A "*" indicates that one or more clauses in the IF part of a rule or a goal clause (G.C.) matches one or more clauses in the THEN part of a rule. The dependency chart is very useful when the knowledge engineer deletes, modifies, or adds rules to the rule base.

Note that in Figure 1, the "*" 's indicate the dependencies for the original rule set. By adding a clause to Rule 2, the "*" dependencies appeared. Note, Rule 2 now references itself—a self-circular rule. By the addition of one clause to Rule 1, the "*"1" dependencies appeared. This also causes the rule set to be circular, since an IF clause of Rule 1 is matched by THEN clauses of Rule 7 and Rule 8 which in turn match an IF clause of Rule 1. Circular rules should be avoided since they can lead to an infinite loop at run time. Some expert systems, such as EKYCIN, handle circular rules in a special way. Nevertheless, the knowledge engineer will want to know which rules are circular. So, CHECK uses the dependency chart to generate graphs representing the interactions between rules, and uses a cyclic graph detection algorithm to detect circular rule chains.

		RULE IDENTIFIERS														
		THEN 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
GOAL AND RULE IDENTIFIERS	IF															
	1							*	*							
	2	*	*													
	3															
	4															
	5															
	6															
	7	*	*													
	8	*	*													
	9	*	*			*	*									
	10	*	*			*	*									
	11							*	*							
	12							*	*							
	13			*	*											
	14			*	*											
15			*	*												
G.C.1															*	
G.C.2															*	
G.C.3													*			
G.C.4													*			
G.C.5										*						
G.C.6									*							
G.C.7									*							

Figure 1

3.0 IMPLEMENTATION OF RULE CHECKER

In solving a problem, the knowledge engineer may write several sets of goal-driven rules with each set having a unique subject category. (In LES it is convenient to put rules in different subject categories so that the system can solve different goals using only those rule sets which apply to that goal.; To solve a particular goal, often he will select several goal-driven rule sets and WHEN rules (demons). Since these rule sets are generated over a period of time, it is quite possible that their interaction will cause some problems. Thus, for each goal it is necessary to compare the rules (in the rule sets specified by that goal) against each other and against the clauses of that goal. We now show an algorithm (in an Algol-like notation) which CHECK uses. The algorithm does the checking for a set of subject categories with N rules and a goal with G clauses.

Because an IF part or a THEN part can have more than one clause, the comparison between one part and another is handled by comparing a clause of one part to every clause in the other part.

These algorithms also work for forward-chaining rules, which are called WHEN rules in LES. However, the criterion unreachable clauses is not applicable to forward-chaining rules.

```

procedure Analyze_KB(Rules,Goal,N,G);
begin
  for i = 1 to N begin
    /* compare rules against each other */
    for j = 1 to N begin
      for k = first_clause(i) to last_clause(i) begin
        for n = first_clause(j) to last_clause(j) begin
          match_result(k) = Compare_clauses(k,n);
        end; /* n */
      end; /* k */
    end; /* j */
  end; /* i */
  /* compare goal clauses against rule clauses */
  for g = 1 to G begin
    for k = first_clause(i) to last_clause(i) begin
      match_result(g) = Compare_clauses(g,k);
    end;
  end;
  /* collect information on attributes coverage */
  for p = first_if_clause(i) to last_if_clause(i) begin
    determine attribute referred by clause p;
    store the attribute's value covered by clause p;
  end;
  /* i */
  /* check for possible problems in rules */
  for l = 1 to N begin
    for k = first_clause(i) to last_clause(i) begin
      matched_rule = Transform(l,k,clause_relations);
      while (matched_rule > 0) do begin
        Check_problems(l,matched_rule,clause_relations);
        matched_rule = Transform(l,k,clause_relations);
      end;
    end;
  end;
  /* check for possible gaps in rules and goals */
  Check_gaps(goal_clauses,rules);
  /* generate the dependency chart */
  Generate_dependency_chart(goal_clauses,rules);
  /* check for possible missing rules */
  for m = first_category to last_category begin
    for n = first_attribute(m) to last_attribute(m) begin
      compare covered values with legal values;
      if not_completely_covered then
        Inform user that some rule is missing;
      if illegal_attribute_value then
        Inform user that attribute value is illegal;
    end; /* for n */
  end; /* for m */
end; /* Analyze_KB */

```

```

procedure Check_gaps(goal_clauses,rules);
begin
  for l = first_then_clause to last_then_clause begin
    if (i did not match any IF clause or GOAL clause) then
      then_result(i) = UNREACHABLE;
    end;
  for j = first_goal_clause to last_goal_clause begin
    if (i not_match any THEN clause & not_askable(i)) then
      goal_result(i) = DEADEND;
    end;
  for i = first_if_clause to last_if_clause begin
    if (i not_match any THEN clause & not_askable(i)) then
      if_result(i) = DEADEND;
    end;
  end; /* Check_gaps */

```

```

procedure Check_problems(i,m,clause_relations);
begin
  conflict_count = 0;
  subset = FALSE;
  superset = TRUE;
  for c = first_if_clause(i) to last_if_clause(i) begin
    if clause_relations(c) = SUBSET then
      subset = TRUE;
    else if clause_relations(c) = SUPERSET then
      superset = TRUE;
    else if clause_relations(c) = CONFLICT then
      conflict_count = conflict_count + 1;
    else
      if_if = SAME; /* composite result may be SAME */
    end;

    if (conflict_count > 1) then
      if_if = DIFFERENT;
    else if (conflict_count=1 & not (subset or superset)) then
      if_if = CONFLICT;
    else begin /* conflict_count = 0 => no conflict */
      if (subset & superset) then
        if_if = DIFFERENT;
      else if subset then
        if_if = SUBSET;
      else if superset then
        if_if = SUPERSET;
      else;
    end;

    if ( (if_if = SAME) &
        number_of_if_clauses(i) = number_of_if_clauses(m) ) then
      for t=first_then_clause(i) to last_then_clause(i) begin
        if (clause_relations(t) = SAME) then
          result(i,m,t) = REDUNDANT;
        else if (clause_relations(t) = CONFLICT) then
          result(i,m,t) = CONFLICT;
        else;
      end;
    else if (if_if = CONFLICT) then
      for t=first_then_clause(i) to last_then_clause(i) begin
        if (clause_relations(t) = SAME) then
          result(i,m,t) = CONFLICT;
        end;
      end;
    else if ( (if_if = SAME) &
              number_of_if_clauses(i) < number_of_if_clauses(m) ) then
      for t=first_then_clause(i) to last_then_clause(i) begin
        if (clause_relations(t) = SAME) then
          result(i,m,t) = SUBSET;
        end;
      end;
    else if ( (if_if = SUBSET) &
              number_of_if_clauses(i) >= number_of_if_clauses(m) ) then
      for t=first_then_clause(i) to last_then_clause(i) begin
        if ( (clause_relations(t) = SAME) or
              (clause_relations(t) = SUBSET) ) then
          result(i,m,t) = SUBSET;
        end;
      end;
    else if ( (if_if = SUPERSET) &
              number_of_if_clauses(i) = number_of_if_clauses(m) ) then
      for t=first_then_clause(i) to last_then_clause(i) begin
        if ( (clause_relations(t) = SAME) or
              (clause_relations(t) = SUPERSET) ) then
          result(i,m,t) = SUPERSET;
        end;
      end;
    else;
  end; /* Check_problems */
end;

```

4.0 gnyMAttv

In this paper we described a program called CHECK whose function is to detect four potential problems (redundant rules, conflicting rules, subsumed rules, and circular rules) and three potential gaps (missing rules, unreachable clauses, and deadend clauses) in a knowledge base utilizing the LES framework. We applied the consistency and completeness verification method of Suwa, Soott, and Shortliffe [4] to the generic expert system LES with good results. Furthermore, we have extended the checking to include circular rules, unreachable clauses, and deadend clauses. We also showed a general algorithm which performs the checking

function efficiently. Finally, as a by-product of the rule checking processing, CHECK generates a dependency chart which shows how the rules couple and interact with each other and with the goals; this chart should help the knowledge engineer to identify immediately the effects of deleting, adding, or modifying rules.

From our experiences with constructing different knowledge bases, we find that many changes and additions to the rule sets occur during the development of a knowledge base. Thus, a tool such as CHECK that can detect many potential problems and gaps in the knowledge base should be very useful to the knowledge engineer in helping him to develop a knowledge base rapidly and accurately.

The major area of improvement for CHECK is the handling of certainty factors in the rules since LES allows the rules to have certainty factors associated with them; this may require the definitions for the seven conditions covered in this paper to be revised.

REFERENCES

- [1] W. A. Perkins, and T. J. Laffey. "LES: A General Expert System and Its Applications", Proc SPIE's Technical Symposium East, Applications of Artificial Intelligence, Arlington, VA., May 3-4, 1984, pp. 46-57.
- [2] w. J. van Melle, System Aids in Constructing Consultation Programs. UMI Research Press, Ann Arbor, MI (1981).
- [3] R. Davis, "Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases", Doctoral dissertation, Computer Science Department, Stanford University, 1976.
- [4] M. Suwa, A. C. Soott, and E. H. Shortliffe, "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System", The AI Magazine. Fall 1982, pp. 16-21.