

# Knowledge Base Verification

**Tin A. Nguyen, Walton A. Perkins,  
Thomas J. Laffey, and Deanne Pecora**

---

*We describe a computer program that implements an algorithm to verify the consistency and completeness of knowledge bases built for the Lockheed expert system (LES) shell. The algorithms described here are not specific to this particular shell and can be applied to many rule-based systems. The computer program, which we call CHECK, combines logical principles as well as specific information about the knowledge representation formalism of LES. The program checks both goal-driven and data-driven rules. CHECK identifies inconsistencies in the knowledge base by looking for redundant rules, conflicting rules, subsumed rules, unnecessary IF conditions, and circular rule chains. Checking for completeness is done by looking for unreferenced attribute values, illegal attribute values, dead-end IF conditions, dead-end goals, and unreachable conclusions. These conditions can be used to suggest missing rules and gaps in the knowledge base. The program also generates a chart that shows the dependencies among the rules. CHECK can help the knowledge engineer detect many programming errors even before the knowledge base testing phase. It also helps detect gaps in the knowledge base that the knowledge engineer and the expert have overlooked. A wide variety of knowledge bases have been analyzed using CHECK.*

---

**L**ayes-Roth (1985) describes several features of rule-based systems that would help to make such a system suitable as a general computing approach. He points out that one of the key features these systems lack is "a suitable verification methodology or a technique for testing the consistency and completeness of a rule set." It is precisely this feature that we address here.

LES is a generic rule-based expert system building tool (Laffey, Perkins, and Nguyen 1986) similar to EMYCIN (Van Melle 1981) that has been used as a framework to construct expert systems in many areas, such as electronic equipment diagnosis, design verification, photointerpretation, and hazard analysis. LES represents factual data in its frame database and heuristic and control knowledge in its production rules. LES allows the knowledge engineer to use both data-driven and goal-driven rules.

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 aid is a syntax checker that examines each rule for syntactic errors such as unbalanced parentheses or misspelled names. Another 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 for existing and potential problems as it is being developed. CHECK analyzes the knowledge base after the rules, facts, and goals have been loaded into LES.

## Related Work

Surprisingly enough, little work has been reported on knowledge base debugging. The TEIRESIAS program (Davis 1976) was the first attempt to automate the knowledge base debugging process. It worked in the context of the MYCIN (Shortliffe 1976) infectious disease consultation system. TEIRESIAS examined the "completed" MYCIN rule set and built rule models showing a number of factors, including which attributes were used to conclude other attributes. Thus, when a new rule was added to MYCIN, it was compared with the rule model for the attributes found in the IF conditions. The program then proposed missing clauses if some attributes found in the IF part of the model did not appear in the new rule. TEIRESIAS did not check the rules as they were initially entered into the knowledge base. Rather, it assumed the knowledge base was "complete" (or close to it), and the knowledge transfer occurred in the setting of a problem-solving session.

Suwa, Scott, and Shortliffe (1982), on which our work is based, wrote a program for verifying knowledge base completeness and consistency. The program was devised and tested within the context of the ONCOCIN system, a rule-based system for clinical oncology.

Unlike TEIRESIAS, ONCOCIN's rule checker is meant to be used as the system is being developed. It examines a rule set as it is read into the system. Knowledge base problems are found by first partitioning the

---

## CHECK identifies inconsistencies in the knowledge base by looking for redundant rules, conflicting rules, subsumed rules, unnecessary IF conditions, and circular rule chains.

---

rules into disjoint sets based upon what attribute is assigned a value in the conclusion. It then makes a table, displaying all possible combinations of attributes used in the IF conditions and the corresponding values that will be concluded in the THEN part of the rule. The table is then checked for conflicts, redundancy, subsumption, and missing rules. Finally, a table is displayed with a summary of any potential errors that were found. The rule checker assumes there should be a rule for each possible combination of values of attributes that appear in the antecedent. It hypothesizes missing rules based on this assumption. Such a process can result in the system hypothesizing rules that have semantically impossible combinations of attributes. Also, if the number of attributes is large, the system can suggest a very large number of missing rules. Nevertheless, the developers of the ONCOCIN system found the rule checker extremely useful in helping them to debug their evolving knowledge base. ONCOCIN uses both data-driven and goal-driven inferencing. Although the rule checker checks the rule set used in the ONCOCIN system, its design is general so that it can be adapted to other rule-based systems.

The intelligent machine model (TIMM) (1985) is an expert system shell that generates its rules from examples (that is, induction). TIMM<sup>TM</sup> has some capability for checking rules. In its method, inconsistency is defined as (1) those rules with the same IF conditions but with different conclusions, (2) those rules with overlapping IF conditions but with different conclusions, and (3) single rules with more than one conclusion.

The first condition is equivalent to logical conflict, but the other two con-


ditions are peculiar to the way the system generalizes its rules. TIMM checks for completeness by searching for points (that is, combinations of attribute values) in the state space that have low similarity to the existing training cases. It does this check by randomly selecting combinations of attributes and finding the situation that is least similar to any training case. This situation is then presented to the user along with a similarity measurement that tells the user how similar the situation is to the closest training case in the knowledge base.

Knowledge engineering system (KES)<sup>TM</sup> (1983) is an expert system shell that has a support tool called INSPECTOR. INSPECTOR identifies all recursive attributes that have been directly or indirectly defined. An example of a recursive attribute is an attribute that occurs in both the antecedent and the consequent of a rule (this definition is similar to that of circular rules, which we discuss later). INSPECTOR can also identify all unattached attributes. An unattached attribute is one that is not contained in the antecedent or conclusion of any rule. However, this situation might not be an error if the knowledge engineer put the attribute in the knowledge base for future use or is using it to contain some type of reference information.

The work described in this article is an extension of the rule-checking program used in the ONCOCIN project. Our work differs from the ONCOCIN effort in that CHECK is applied to the entire set of rules for a goal, not just the subsets which determine the value of each attribute. Because of this global view of the knowledge base, CHECK includes several new rule-checking criteria, including unreachable conclusions, dead-end IF conditions, dead-end goals, unnecessary IF conditions, unreferenced attribute values, and ille-

gal attribute values. Furthermore, CHECK produces dependency charts and detects any circular rule chains. This rule-checking system was devised and tested on a wide variety of knowledge bases built with a generic expert system shell rather than on a single knowledge base as in the ONCOCIN project.

### Potential Problems in the Knowledge Base

 static analysis of the rules can detect many potential problems that exist in a knowledge base. First, we identify knowledge base problems that can be detected by performing an analysis of goal-driven rules and then give definitions and examples of such problems. Later in this article, we look at how these definitions must be modified for data-driven rules.

Knowledge base problems can only be detected if the rule syntax is restrictive enough to allow one to examine two rules and determine whether situations exist in which both can succeed and whether the results of applying the two rules are the same, conflicting, or unrelated. In rule languages that allow an unrestricted syntax, it is difficult or impossible to implement the algorithms described in this article.

### Checking for Consistency

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, unnecessary IF conditions, and circular-rule chains. These five potential problems are defined in the subsections that follow.

**Redundant Rules** Two rules are redundant if they succeed in the same situa-

tion and have the same conclusions. In LES this statement means that the IF parts of the two rules are equivalent, and one or more conclusions are also equivalent. The IF parts of two rules can be equivalent only if each part has the same number of conditions, and each condition in one part is equivalent to a condition in the other part. Because LES allows variables in rules, two conditions are equivalent if they are unifiable.

Formally, with the notation from predicate calculus, rule  $p(x) \rightarrow q(x)$  is equivalent to the rule  $p(y) \rightarrow q(y)$ , where  $x$  and  $y$  are variables, and  $p$  and  $q$  are logical relationships.

For example, consider the two rules that follow:

```
IF      ?X has a hoarse cough, AND
        ?X has difficulty breathing
THEN   type-of-disease of ?X is
        CROUP

IF      ?Y has difficulty breathing,
        AND
        ?Y has a hoarse cough
THEN   type-of-disease of ?Y is
        CROUP
```

?X and ?Y represent variables that will be instantiated to a person in the database. These two rules would be redundant even if they used different variables and their IF conditions were in a different order.

As reported by Suwa, Scott, and Shortliffe (1982), redundancy in a knowledge base does not necessarily cause logical problems, although it might affect efficiency. In a system where the first successful rule is the only one to succeed, a problem will arise only if one of two redundant rules is revised or deleted, and the other is left unchanged. Also, unless the system uses some type of scoring scheme (for example, certainty factors), redundancy should not cause a problem.

**Conflicting Rules** Two rules are conflicting if they succeed in the same situation but with conflicting conclusions. In LES this statement means that the IF parts of the two rules are equivalent, but one or more conclusions are contradictory.

Formally, with the notation from predicate calculus, the rule  $p(x) \rightarrow \text{not}(q(x))$  is contradictory to the rule

$p(x) \rightarrow q(x)$ .

For example, consider the two rules that follow:

```
IF      ?X has a hoarse cough, AND
        ?X has difficulty breathing
THEN   type-of-disease of ?X is
        CROUP

IF      ?X has a hoarse cough, AND
        ?X has difficulty breathing
THEN   type-of-disease of ?X is
        BRONCHITIS
```

These two rules are conflicting (assuming the attribute type-of-disease is single-valued) because given the same information, one rule concludes that the disease is croup, and the other concludes bronchitis.

#### NOTE

It is possible that rules with similar premises might not conflict at all, especially when they are concluding values for a multivalued attribute. (A multivalued attribute can assume multiple values simultaneously. For example, a person can be allergic to many different drugs or can be infected by numerous organisms.)

**Subsumed Rules** One rule is subsumed by another if the two rules have the same conclusions, but one contains additional constraints on the situations in which it will succeed. In LES this statement means one or more conclusions are equivalent, but the IF part of one rule contains fewer constraints or conditions than the IF part of the other rule.

Formally, with the notation from predicate calculus, the rule  $(p(x) \text{ and } q(y)) \rightarrow r(z)$  is subsumed by the rule  $p(x) \rightarrow r(z)$ . Whenever the more restrictive rule succeeds, the less restrictive rule also succeeds, resulting in redundancy.

For example, consider the two rules that follow:

```
IF      ?X has flat pink spots on his
        skin, AND
        ?X has a fever
THEN   type-of-disease of ?X is
        MEASLES

IF      ?X has flat pink spots on his
        skin
THEN   type-of-disease of ?X is
        MEASLES
```

In this case, we would say that rule 1 is subsumed by rule 2 because rule 2 only needs a single piece of information to conclude "measles." Whenever rule 1 succeeds, rule 2 also succeeds.

**Unnecessary IF Conditions** Two rules contain unnecessary IF conditions if the rules have the same conclusions, an IF condition in one rule is in conflict with an IF condition in the other rule, and all other IF conditions in the two rules are equivalent. With our notation from predicate calculus, if we have the rule  $(p(x) \text{ and } q(y)) \rightarrow r(z)$  and the rule  $(p(x) \text{ and } \text{not}(q(y))) \rightarrow r(z)$ , the condition involving  $q(y)$  in each rule is unnecessary. These two rules could be combined into  $(p(x) \text{ and } (q(y) \text{ or } \text{not}(q(y)))) \rightarrow r(z)$ . The condition  $(q(y) \text{ or } \text{not}(q(y)))$  resolves to TRUE; thus, the rule becomes  $p(x) \rightarrow r(z)$ . In this case, the unnecessary IF condition actually indicates that only one rule is necessary.

For example, consider the two rules that follow:

```
IF      ?X has flat pink spots on his
        skin, AND
        ?X has a fever
THEN   type-of-disease of ?X is
        MEASLES

IF      ?X has flat pink spots on his
        skin
        ?X does not have a fever
THEN   type-of-disease of ?X is
        MEASLES
```

In this case, the second IF condition in each rule is unnecessary. Thus, the two rules could be collapsed into one.

A special case occurs when two rules have the same conclusion, one rule containing a single IF condition that is in conflict with an IF condition of the other rule which has two or more IF conditions. With our notation from predicate calculus, if we have the rule  $(p(x) \text{ and } q(y)) \rightarrow r(z)$  and the rule  $\text{not}(q(y)) \rightarrow r(z)$ , then the second IF condition in the first rule is unnecessary, but both rules are still needed and can be reduced to  $(p(x) \rightarrow r(z) \text{ and } \text{not}(q(y)) \rightarrow r(z))$ .

**Circular Rules** A set of rules is circular if the chaining of these rules in the set forms a cycle. With our notation from predicate calculus, if we have the set of rules  $p(x) \rightarrow q(x)$ ,  $q(x) \rightarrow r(x)$ ,

and  $r(x) \rightarrow p(x)$ , and the goal is  $r(A)$ , where  $A$  is a constant, then the system enters an infinite loop at run time unless the system has a special way of handling circular rules. Also, this definition includes the possibility of a single rule forming a circular cycle (for example,  $p(x) \rightarrow p(x)$ ).

For example, consider the following set of rules:

```
IF      temperature of ?X > 100 (in
        Fahrenheit)
THEN   ?X has a fever
IF      ?X has a fever, AND
        ?X has flat pink spots on his
        skin
THEN   type-of-disease of ?X is
        MEASLES
IF      type-of-disease of ?X is
        MEASLES
THEN   temperature of ?X > 100 (in
        Fahrenheit)
Given a goal of
        type-of-disease of patient is
        MEASLES,
```

this set of rules would go into an infinite loop if one attempted to backward chain them together because the goal would match the conclusion of rule 2, the first IF condition of rule 2 would match the conclusion of rule 1, the IF condition of rule 1 would match the conclusion of rule 3, and the IF part of rule 3 would match the conclusion of rule 2, thus completing our circular chain.

## Checking for Completeness

The development of a knowledge-based system is an iterative process in which knowledge is encoded, tested, added, changed, and refined. Knowledge flows from the expert into the knowledge base by way of a middleman (the knowledge engineer). This iterative process often leaves gaps in the knowledge base that both the knowledge engineer and the expert have overlooked during the knowledge-acquisition process. Furthermore, as the number of rules grows large, it becomes impossible to check every possible path through the system. In our research, we have found four situations indicative of gaps (that is, missing rules) in the knowledge base: (1) unreferenced attribute values, (2) dead-

## The TEIRESIAS program (Davis 1976) was the first attempt to automate the knowledge base debugging process.

end goals, (3) unreachable conclusions, and (4) dead-end IF conditions. Any one of these four conditions might indicate that there is a rule missing.

In the ONCOCIN system, the rule checker assumes there should be a rule for each possible combination of values of attributes that appear in the antecedent. In practice, we found this criterion causes the system to hypothesize a very large number of missing rules and chose to leave it out of our checking process. This problem was not serious in the ONCOCIN project because the checker was only tested on a single application.

LES (and EMYCIN) allows the knowledge engineer the feature of strong typing the defined attributes, thus facilitating the detection of gaps. For each attribute, one can define a set of properties for it, including whether the user can be queried for the value, and a set of values the attribute can take on (that is, its legal values). This method has long been recognized in software engineering as an excellent programming practice. In fact, the newer programming languages (for example, Pascal and Ada) have type-checking capabilities along these lines.

LES allows the knowledge engineer to define properties about each slot in its factual database, including the set or range of acceptable attribute values, system ability to query the user for the attribute, and the attribute's type (single valued or multivalued). In the subsections that follow, we describe how LES uses these properties to aid it in finding gaps and errors in the knowledge base.

**Unreferenced Attribute Values** Unreferenced attribute values occur when some values in the set of possible values of an object's attribute are not covered by any rule's IF conditions. In other words, 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 can cause it to make a wrong conclusion when an uncovered attribute value is encountered at run time. Unreferenced attribute values might also indicate that rules are missing.

For example, suppose we have the attribute TEMPERATURE with the set of legal values {high, normal, low}. If the attribute values high and normal are used in the IF conditions of rules but not low, CHECK alerts the knowledge engineer that low is not used. The knowledge engineer would then have to decide if a rule is missing or if the value low should be removed from the set of legal values.

**Illegal Attribute Values** An illegal attribute value occurs when a rule refers to an attribute value that is not in the set of legal values. This error is often caused by a spelling mistake. No extra work is required to check for this condition because it is a by-product of checking for unreferenced attribute values.

Suppose we have the attribute TEMPERATURE with the set of legal values {high, normal, low}. If a rule has a condition such as

```
IF temperature of ?X is very high... or
...THEN temperature of ?X is medium
```

CHECK alerts the knowledge engineer that the values "very high" and "medium" are illegal attribute values for temperature.

**Unreachable Conclusions** In a goal-driven production system, the conclusion of a rule should either match a goal or match an IF condition of another rule (in the same rule set). If there are no matches for the conclusion, it is unreachable.

For example, suppose we have the following rule:

```
IF      temperature of ?X > 100 (in
        Fahrenheit)
THEN   ?X has a fever
```

If the condition ?X has a fever does not appear in the IF part of any rule and is not part of the goal, Check alerts the knowledge engineer that this conclusion is unreachable.

It is possible that such a rule is merely extraneous, in which case it

might affect efficiency but not the outcome because it will never be triggered. It is also possible that the conclusion does not match a goal (or subgoal) because of a terminology error. For example, a rule might exist with an IF condition of the form

IF       ?X has an elevated temperature

THEN ... ,

where the terms "elevated temperature" and "fever" are synonymous to the expert but not to the expert system.

**Dead-End IF Conditions and Dead-End Goals** To achieve a goal (or subgoal) in a goal-driven system, either the attributes of the goal must be askable (user provides needed information), or the goal must be matched by a conclusion of one of the rules in the rule sets applying to the goal. If neither of these requirements is satisfied, then the goal cannot be achieved (that is, it is a dead-end goal). Similarly, the IF conditions of a rule must also meet one of these two conditions, or they are dead-end conditions.

For example, suppose we have the following as a goal (or subgoal):

type-of-disease of patient is  
MEASLES ,

If the attribute type-of-disease is not askable, and there are no rules that conclude this fact, then this goal would be labeled dead-end goal.

## Dependency Chart and Circular-Rule-Chain Detection

As a by-product of rule checking, CHECK generates two dependency charts. One chart shows the interactions among the data-driven rules, and the other shows the interactions among the goal-driven rules and the goals. An example of a dependency chart for a small set of rules is shown in Figure 1.

An \* indicates that one or more IF conditions or a goal condition (GC) matches one or more conclusions of a rule. The dependency chart is useful when the knowledge engineer deletes, modifies, or adds rules to the rule base because it is a means of immediately

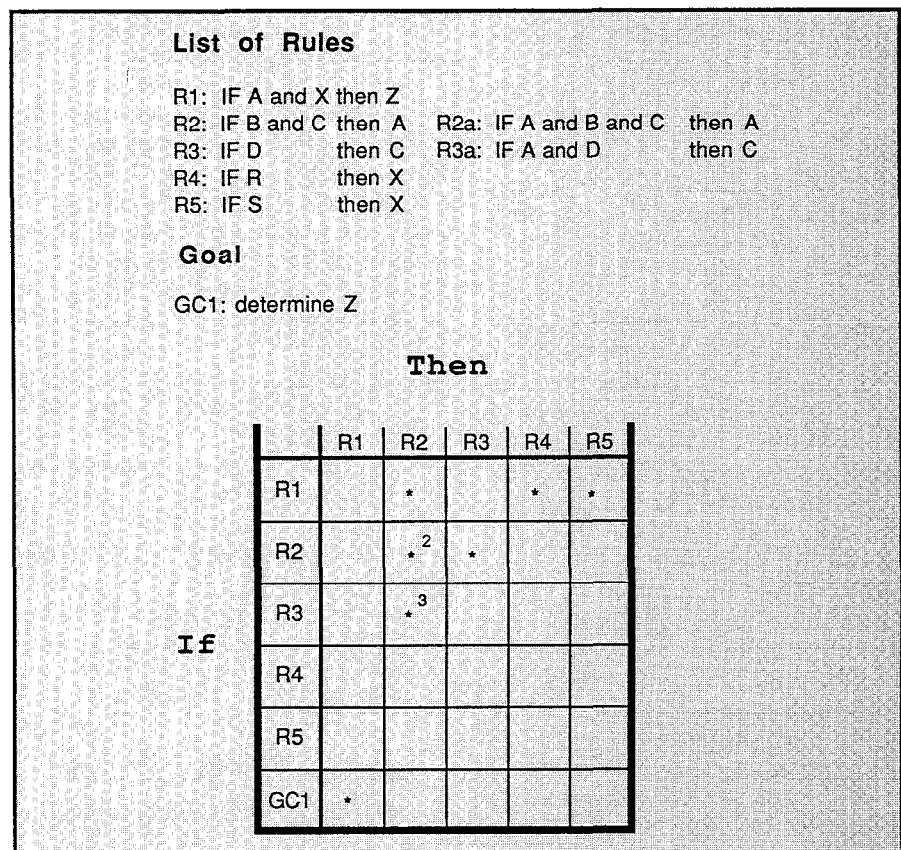


Figure 1. A Simple Rule Set and Its Dependency Chart.

seeing the dependencies among the rules.

Note that in figure 1 the asterisks (\*) indicate the dependencies for the original rule set. (For example, the \* in row R1, column R2 indicates that an IF clause of rule R1 is concluded by rule R2). Adding a condition to rule R2 (see rule R2a) caused the \*<sup>2</sup> dependency to appear. Note that rule R2a now references itself (that is, it is a self-circular rule). The addition of one condition to rule R3 (see rule R3a) caused the \*<sup>3</sup> dependency to appear. This addition also causes the rule set to be circular because a condition of rule R3a is matched by the conclusion of rule R2, and a condition of rule R2 matches the conclusion of rule R3a.

Circular rules should be avoided because they can lead to an infinite loop at run time. Some expert systems, such as EMYCIN, handle circular rules in a special way. Nevertheless, the knowledge engineer will want to know which rules are circular. 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.

## Checking Data-Driven Rules

To this point, we have only considered goal-driven rules, but as discussed earlier, LES also supports data-driven inferencing. The data-driven rules are called WHEN rules. A WHEN rule consists of one or more WHEN conditions (similar to IF conditions) and one or more conclusions.

Checking a data-driven rule set for consistency and completeness is very similar to checking goal-driven rules. The detection of conflicting rules, redundant rules, subsumed rules, circular-rule chains, dead-end IF conditions, unreferenced attribute values, and illegal attribute values is done in the same manner as described earlier for checking goal-driven rules. However, the detection of unreachable conclusions is not applicable in checking a data-driven rule set because there are no goals to match to the conclusions.

## Implementation of the Rule Checker

**I**n solving a problem with LES, the knowledge engineer partitions the rules into sets, where each set is associated with a subject category. (In LES one can have multiple goals, and each goal has zero or more rule sets associated with it.) The set of rules for each goal can then be checked independently. To check a set of rules, the program performs five steps, as described in the following paragraphs (note that the phrase IF part of a rule means the entire set of antecedent clauses, and the phrase THEN part of rule means the entire set of conclusions).

First, each IF and THEN clause of every rule in the set (and each goal clause) is compared against the IF and THEN clauses of every other rule in the set. The comparison of one clause against another results in a label of SAME, DIFFERENT, CONFLICT, SUBSET, or SUPERSET being stored in a two-dimensional (2-D) table maintaining the interclause relationships. The comparison operation is not straightforward because variables and the ordering of clauses must be taken into consideration.

Second, the IF part and the THEN part of every rule (and the goal) are compared against the IF and THEN part of every other rule to deduce the relationships. This process is done using the 2-D table of interclause relationships, together with the number of clauses in each part, to determine how an IF or THEN part (or goal) is related to another IF or THEN part. The possible relationships resulting from the deductions are the same as described in the first step.

Third, the part relationships of each rule are compared against the part relationships of every other rule to deduce the relationships among the rules. These comparisons are then output to the user; the possible relationships are SAME (redundant), CONFLICT, SUBSET (subsumption), SUPERSET (subsumption), UNNECESSARY CLAUSES, or DIFFERENT.

Fourth, gaps are checked for using the 2-D table of interclause relation-

ships. Unreachable conclusions are identified by finding those THEN clauses which have the DIFFERENT relationship for all IF clauses and goals. Dead-end goals and IF conditions are identified when the DIFFERENT relationship exists for all conclusions, and the attribute these goals and conditions refer to is not askable.

Fifth, the dependency chart is also generated from the 2-D table of interclause relationships. A rule is said to be dependent on another rule if any of its IF conditions have the relationship SAME, SUBSET, or SUPERSET with any of the other rules' conclusions. The actual algorithms that used to do the checking appear in Nguyen, et al. (1985). (Since the publication of these algorithms, we have added the capability to check for unnecessary IF conditions. We have also revised our definition of missing rules.)

## How Certainty Factors Affect the Checking

LES allows the use of certainty factors in its goal-driven and data-driven rules. Certainty factors are implemented in the same manner as in the EMYCIN system, with a value of +1.0 for meaning definitely true, 0.0 for unknown, and -1.0 for definitely false. The presence of certainty factors further complicates the process of checking a knowledge base. Allowing rules to conclude with less than certainty and allowing data to be entered with an associated certainty factor affects our definitions, as shown in the following paragraphs.

A conflict--when two rules succeed in the same situation but with different conclusions--is a common occurrence in rule sets using certainty factors. Often, given the same set of symptoms, the expert might wish to conclude different values with different certainty factors.

In reference to redundancy, rules that are redundant can lead to serious problems. They might cause the same information to be counted twice, leading to erroneous increases in the weight of their conclusions.

Subsumption is used quite often in

rule sets with certainty factors. The knowledge engineer frequently writes rules so that the more restrictive rules add weight to the conclusions made by the less restrictive rules.

In regard to Unnecessary IF conditions, IF conditions that are labeled as unnecessary when rules conclude with absolute certainty might be necessary when dealing with certainty factors. The knowledge engineer might wish to conclude a value at different certainty factors. If the rules conclude with the same certainty factor, then the IF conditions are still unnecessary.

Certainty factors do not affect our definition of or the way we detect unreferenced attribute values. The same is true with our definition of illegal attribute values.

Finding dead-end IF conditions (or dead-end goals) becomes complex with certainty factors. LES, like EMYCIN, allows the user to specify a threshold at which point the value becomes unknown. (In MYCIN this threshold is set at 0.2.) Thus, a dead-end goal could occur if there is a THEN clause that concludes with a certainty factor less than the threshold (or a chain of rules that when combined produces a certainty factor less than the threshold). For example, suppose there is a linear reasoning path of three rules (R1, R2, and R3), where A is to be asked of the user and D is the initial goal which initiated this line of reasoning

	R1	R2	R3
A ----->	B----->	C----->	
	0.4	0.7	0.7

If A is known with certainty, D would only be known with a certainty factor of  $(0.4)(0.7)(0.7) = 0.19$ . This factor is less than the threshold used in MYCIN, and, thus, D would be a dead-end goal. If D is an IF condition rather than a goal, then D would be a dead-end condition if it were not askable, and there were no other lines of reasoning to determine it.

Detecting unreachable conclusions in a rule set with certainty factors also becomes complex. A conclusion in a rule could be unreachable even though its IF part matches a conclusion in another rule. This situation might occur if the conclusion that matches one of the IF conditions cannot be determined with a certainty factor

above the threshold. For example, suppose we have the following two rules:

R1: IF A THEN B (certainty factor = 0.1)

R2: IF B THEN C (certainty factor = 1.0)

If the only way to determine B was with rule R1, then the conclusion of rule R2 would be unreachable because even if A were known with certainty, C could not be determined with a certainty factor above the threshold of 0.2.

The detection of circular-rule chains is not affected by certainty factors. However, it should be noted that certainty factors might cause a circular chain of rules to be "broken" if the certainty factor of a conclusion falls below the threshold of 0.2.

From this discussion, we can see that certainty factors only introduce minor extensions to the checking process, but the knowledge engineer should be aware of these differences. (The program CHECK does not look at certainty factors when checking a knowledge base.)

## Summary

In this article we described CHECK, a program whose function is to check a knowledge base for consistency and completeness. The program detects several potential problems, including redundant rules, conflicting rules, subsumed rules, unnecessary IF conditions, and circular rules. CHECK also attempts to verify completeness in the knowledge base by looking for potential gaps, including unreferenced attribute values, illegal attribute values, missing rules, unreachable conclusions, dead-end IF conditions, and dead-end goals. We extended and applied the verification method for consistency and completeness of Suwa, Scott, and Shortliffe (1982) to a variety of knowledge bases built with the generic expert system shell LES with excellent results. We showed a general algorithm that efficiently performs the checking function in a single pass through the rules. We also built a version of CHECK that works with knowledge bases built with the Automated Reasoning Tool [ART<sup>sup</sup>/TM] (Nguyen 1987).

Finally, as a by-product of the rule-checking process, CHECK generates a dependency chart that shows how the rules couple and interact with each other and the goals. These charts can help the knowledge engineer visualize the effects of deleting, adding, or modifying rules.

From our experiences with constructing different knowledge bases, we found 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 to develop a knowledge base rapidly and accurately.

As the field of knowledge-based systems matures, large expert systems will be fielded in critical situations. Because it will be impossible to test all paths beforehand, one must be assured that deadly traps such as circular rules and dead-end clauses do not exist in the knowledge base. Thus, a checking capability similar to the one described in this article is essential.

## References

- Davis, R. 1976. Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases. Ph D. diss., Dept. of Computer Science, Stanford Univ.
- Hayes-Roth, F. 1985. Rule-Based Systems. *Communications of the ACM* 28(9): 921-932.
- KES General Description Manual. 1983. Software Architecture and Engineering, Inc., Arlington, Va., p. 33.
- Laffey, T. J.; Perkins, W. A.; and Nguyen, T. A. 1986. Reasoning about Fault Diagnosis with LES. *IEEE Expert, Intelligent Systems and Their Applications* 1(1): 13-20.
- Nguyen, T. A. 1987. Verifying Consistency of Production Systems. In *Proceedings of the Third Conference on Artificial Intelligence Applications*, 4-8. Washington D.C.: IEEE Computer Society Press.
- Nguyen, T. A.; Perkins, W. A.; Laffey, T. J.; and Pecora, D. 1985. Checking an Expert System's Knowledge Base for Consistency and Completeness. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 374-378. Menlo Park, Calif.: American Association for Artificial Intelligence.

Shortliffe, E. H. 1976. *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier.

Suwa, M.; Scott, A. C.; and Shortliffe, E. H. 1982. An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System. *AI Magazine* 3(4): 16-21.

TIMM User's Manual. 1985. General Research Corporation, Santa Barbara, Calif., pp. 46-49..\*

Van Melle, W. J. 1981. *System Aids in Constructing Consultation Programs*. Ann Arbor, Mich.: UMI Research Press.

## Note

\*TIMM™ is a registered trademark of General Research Corporation.

\*ART is a registered trademark of Inference Corporation.

## ■ Nonmonotonic ■ Reasoning ■ Workshop

### *Proceedings of the Workshop*

Sponsored by  
the American Association for  
Artificial Intelligence  
17-19 October 1984  
Mohonk Mountain House  
New Paltz, New York

The Proceedings of this successful AAAI Workshop on Nonmonotonic Reasoning are still available in limited quantities. Covering a variety of topics, papers in the proceedings emphasize formal approaches to nonmonotonicity, with circumscription, default and auto-epistemic reasoning, being the favorite topics.

**\$20.00 postpaid.**

To order, send a check or money order to:

**Publications Department  
American Association for  
Artificial Intelligence  
445 Burgess Drive  
Menlo Park, California 94025**

For free information, circle no. 49