UVT: A Unification-Based Tool for Knowledge Base Verification

Faruk Polat, University of Minnesota
H. Altay Guvenir, Bilkent University

THIS METHOD CHECKS KNOWLEDGE BASE SYSTEMS FOR COMPLETENESS AND CONSISTENCY BY GENERATING INFERRED RULES AND THEN CONSIDERING THEIR EFFECTS IN A VERIFICATION PROCESS.

ONE OF THE GREATEST DIFFICULTIES in developing an expert system is knowledge acquisition, the process of building the knowledge base. A knowledge base might be incomplete or inconsistent from the start, since human experts are not prepared to provide all the knowledge needed in one complete and consistent chunk. There might be cases that the expert has not considered, or items (rules, for example) that need to be rephrased. Even after the knowledge base has been built, its maintenance usually requires a complete validation if any item is modified, removed, or added. Therefore, validation is considered a mandatory step in developing knowledge-based systems.[1]

An expert system cannot be tested, even on simple cases, until much of its knowledge base is encoded. Regardless of how an expert system is developed, its builders can profit from a systematic check of the knowledge base without having to gather extensive data for test runs, even before the full reasoning mechanism is functioning. This verification can be achieved by developing a program to check the knowledge base for consistency and completeness.[2]

Our proposed method for verifying knowledge bases is based on the unification of rules.[3] One characteristic that distinguishes our approach from other verification tools is that it infers some of the rules that are not explicitly given in the rule base and considers their effect in the verification process. Our method can determine conflicting, redundant, subsumed, circular, and dead-end rules; redundant If conditions in rules; and cycles and contradictions within rules. We have implemented our method in a computer program called UVT (for unification-based verification tool) and tested it on sample knowledge bases.

Rules for knowledge representation

When planning an expert system, developers must decide on a knowledge representation scheme that is most suitable to the application. We chose a rule-based representation scheme because of the modularity it provides and the simple, uniform interpretive procedure that is often sufficient in rule-based systems. Rule-based representation is also easy to learn and use.

Our method assumes that the knowledge base rules have only one literal in their consequents and a conjunction of literals in their antecedents. A literal is either a predicate or the negation of a predicate. A predicate has a name and a finite number of arguments, which can be variables or constants. Although most expert systems use certainty factors associated with rules to handle uncertainty, our method does not depend on certainty factors. Our verification method is also independent of the inference mechanism to be used with the

0885/9000/93/0600-0069 $3.00

An earlier version of this article appeared on pp. 147-163 of the *Proceedings of the European Workshop on the Verification and Validation of Knowledge-Based Systems (EuroVAV '91)*, published by Logica Cambridge, Cambridge, England, 1991.

Example	Pairs of conjunctions
1	connected (?device-x, Living-room-outlet) & works (?device-x) connected (Iron, Living-room-outlet) & works (Iron)
2	connected (?device-x, Living-room-outlet) & works (?device-x) connected (?device-y, Living-room-outlet) & works (?device-y)
3	connected (?device-x, Living-room-outlet) & works (?device-x) works (?device-y) & connected (?device-y, Living-room-outlet)
4	connected (?device-x, Living-room-outlet) & works (?device-y) works (?device-z) & connected (?device-t, Living-room-outlet)

Figure 1. Conjunctions that might occur as the antecedents of rules in the domain of the diagnosis of household electrical devices and wiring systems.

Table 1. Relationships between the example conjunctions in knowledge bases KB_1 and KB_2.

	KB_1		KB_2	
Example	Instantiation of variables	Equivalent	Instantiation of variables	Equivalent
1	?device-x = Iron	Yes	?device-x = Lamp	No
2	?device-x = Iron, ?device-y = Iron	Yes	?device-x = Lamp, ?device-y = Lamp	Yes
3	?device-x = Iron, ?device-y = Iron	Yes	?device-x = Lamp, ?device-y = Iron	No
4	?device-x = Iron, ?device-y = Iron, ?device-t = Iron, ?device-z = Iron	Yes	?device-x = Lamp, ?device-y = Iron, ?device-t = Lamp, ?device-z = Iron	Yes

rule base. However, as we show later, some types of problems are specific to the inference mechanism used.

In this article, variables start with a question mark, predicate names start with lower case letters, and constants are capitalized and may be of type integer, real, or string. For example, the following fact could appear in an expert system designed to diagnose household electrical devices and wiring systems:

```
temperature (Heater, High)
```

This fact states that the temperature of the heater is high. An example rule is

```
if type (?device, TV-set) &
   temperature (?device, High) &
   ~ works (TV-set)
then state (?device, Broken) ;
```

where *?device* is a variable, "TV-set," "High," and "Broken" are constants, "type," "temperature," "works," and "state" are predicate names, "&" means "and," and the symbol "~" denotes the negation of a predicate. This rule states that if there is a TV set whose temperature is high and it is not functioning, we can conclude this device is broken.

Unification

Knowledge base rules with common predicates might be related. During verification, UVT compares the literals to determine the relationships between them. These common predicates might be equivalent even though they are not exactly equal. The predicates in conjunctions can be in any order, although there might be some restrictions imposed by a substitution list. To determine the equivalence of common predicates, we use a technique called unification.[4-5]

The conjunctions in Figure 1 might occur as the antecedents of rules in the example electrical-device domain mentioned earlier. In the first set of conjunctions, suppose there is no restriction imposed by a substitution list; that is, the variable *?device-x* in the first conjunction has not yet been instantiated. We can easily see that if variable *?device-x* takes the value Iron, these two conjunctions are equivalent; however, we cannot be sure whether the inference engine will instantiate the variable *?device-x* to the constant Iron.

In the second example in Figure 1, we can make a definite judgment about the conjunctions' equivalence, because *?device-x* can be unified with *?device-y* (assuming both are uninstantiated) before the matching begins. The conjunctions in the third example pair are not definitely equivalent, because the order of predicates is different and the constants that will be provided by the inference engine for variables *?device-x* and *?device-y* might not always be the same.

Although the order of predicates in the fourth pair is different, the conclusion about the conjunctions' equivalence is definite assuming the four variables are uninstantiated before matching begins.

Let's assume that the inference engine matches predicates with knowledge base facts from top to bottom. To see why the equivalence of two conjunctions of predicates might not be definite, consider the following sets of facts in knowledge bases KB_1 and KB_2:

```
KB1:
connected (Iron, Living-room-outlet)
works (Iron)
connected (Lamp, Living-room-outlet)
works (Lamp)

KB2:
connected (Lamp, Living-room-outlet)
connected (Iron, Living-room-outlet)
works (Iron)
works (Lamp)
```

Table 1 compares the relationships between the conjunctions in Figure 1, taking into account the facts in KB_1 and KB_2.

This example shows that the ordering of data can affect the instantiations to the variables. The conjunctions in examples 2 and 4 are equivalent no matter in what order the facts are derived. On the other hand, the conjunctions in examples 1 and 3 are not equivalent in the case of KB_2, although they are in the first knowledge base. This leads to uncertainties in detecting some types of knowledge base anomalies.

Inferred rules

Before testing, we must find the rules that are inferred by the knowledge base, because they might contradict other rules or cause circular chains. However, computing all the inferred rules is infeasible; there are just too many.

We use the term "inferred rules" for the rules that can be obtained using the transitivity property of rules. We find them by computing the transitive closure of the rules. A new rule can be inferred from two rules if the literal in the consequent of one rule is unifiable with a literal in the antecedent of another. In that case, the inferred rule's antecedent will be the conjunction of the initial rules' antecedents, excluding the literal that causes transitivity, and the consequent will be the second rule's consequent after applying the substitution obtained by the unification. For example, given

```
R40: if type (?device-x, Heating) &
        temperature (?device,x, High)
     then getspower (?device-X)

R41: if getspower (?appliance) &
        ~works (?appliance)
     then needs-diagnosis (?appliance)
```

we can infer

```
Rnew:  if type (?appliance, Heating) &
          temperature (?appliance, High) &
          ~works (?appliance)
       then needs-diagnosis (?appliance)
```

In forming an inferred rule's antecedent, UVT eliminates all the duplicate literals in the antecedent. In case two literals cause a tautology, UVT cannot infer a new rule, but informs the knowledge engineer about the potential problem that might be caused by the two rules. If the rules have certainty factors, UVT computes the inferred rules' certainty factors and then discards the rules whose certainty factor is below a given threshold.

Suppose we have the following set of rules from the domain of household electrical devices:

```
R1: if type (?device, Iron) &
       temperature (?device, High)
    then works (?device)

R2: if works (?device)
    then state (?device, OK)

R3: if state (?device, OK)
    then ~problem (?device)

R4: if type (?device, Iron) &
       temperature (?device, High)
    then problem (?device)
```

CONSISTENCY CHECKING TESTS WHETHER A SYSTEM PRODUCES SIMILAR ANSWERS TO SIMILAR QUESTIONS. COMPLETENESS CHECKING TESTS WHETHER IT ANSWERS ALL REASONABLE SITUATIONS IN ITS DOMAIN OF EXPERTISE.

Using transitivity, we can infer

```
I1: if type (?device, Iron) &
       temperature (?device, High)
    then state (?device, OK)

I2: if works (?device)
    then ~problem (?device)

I3: if type (?device, Iron) &
       temperature (?device, High)
    then ~problem (?device)
```

I1 is inferred from R1 and R2, I2 from R2 and R3, and I3 from R1, R2, and R3. Without considering the inferred rules, it is hard to see that the original set of rules contains a contradiction. However, it is clear that R4 and the inferred rule I3 conflict.

To find the inferred rules, UVT uses a recursive algorithm[3] to compute the transitive closure of the initial rule base. The worst-case time complexity of the algorithm is $O(N^3)$, where N is the number of rules in the rule base. Most rule-based systems do not have long rule chains; instead, they have many short chains.

The knowledge base problems detectable by UVT

After finding and appending the inferred rules to the knowledge base, UVT checks the rules against two requirements: consistency and completeness. Consistency checking includes testing whether the system produces similar answers to similar questions. Inconsistencies in a knowledge base might appear as conflicts, redundancies, or subsumptions. Two rules conflict with each other if they succeed in the same situation but result in contrary conclusions. They are redundant if they succeed in the same situations and result in the same conclusions. A rule is subsumed by another rule if it has the same conclusions but only a portion of the antecedents of the other rule as its antecedents. Completeness checking includes testing whether the system answers all reasonable situations within its domain of expertise. When a system is complete, everything that can be derived from the data is derived. Completeness can be achieved by identifying and removing knowledge gaps in a knowledge base.

In this discussion, we'll ignore certainty factors and assume that the inference mechanism processes the conditions in a rule's antecedent from left to right.

Redundant rules. Redundant rules are those that succeed in the same situation and have the same result. In other words, when two rules' antecedents are equivalent, their consequents are also equivalent. Two antecedents are equivalent when they can be unified and have an equal number of literals; two consequents are equivalent if they can be unified. For example, consider

```
R5: if connected (?device, ?point) &
       works (?device)
    then hasPower(?point)

R6: if connected (?appliance,
       ?plug_out) &
       works (?appliance)
    then hasPower(?plug_out)
```

R5 and R6 are redundant no matter which inference mechanism (backward or forward chaining) is used, because the number of literals in the antecedents is equal, and the antecedents and the consequents are unifiable with substitutions (*?appliance/?device, ?plug_out/?point*) to the first

Related work

The first attempt to automate knowledge base debugging was the Teiresias[1,2] program, developed in the context of the Mycin medical expert system.[1] Teiresias is an interactive program that lets an expert judge whether a Mycin diagnosis is correct, track down the errors in the knowledge base that led to inconsistent conclusions, and alter, delete, or add rules in order to fix these errors.

The Rule Checker Program[3] for verifying consistency and completeness was developed to be used with Oncocin,[1,3] a rule-based expert system for clinical oncology. RCP determines completeness and missing rules through combinatorial enumeration. It hypothesizes missing rules by assuming there is a rule for each possible combination of values of attributes that appear in the antecedent.

Check[4] is a knowledge base verification tool for LES, Lockheed's rule-based expert system shell. Check assumes that rules are naturally separated by subject categories. It identifies inconsistencies in the knowledge base by looking for redundant, conflicting, and subsumed rules, unnecessary If conditions, and circular rule chains. The program uses dependency charts to detect rule chains.

The Expert System Checker[5] is a decision-table-based checker for rule-based systems. ESC first constructs a master decision table for the entire knowledge base, then automatically splits it into subtables, checks each subtable for completeness and consistency, and reports missing rules.

Preece, Shinghal, and Batarekh surveyed the work on verification of expert-system knowledge bases that are based on first-order logic.[6] Their overview compares five verification programs, including RCP and Check.

References

1. B.G. Buchanan and E.H. Shortliffe, *Rule-Based Expert Systems*, Addison-Wesley, Reading, Mass., 1985.
2. R. Davis, "Interactive Transfer of Expertise: Acquisition of New Inference Rules," *Artificial Intelligence*, Vol. 12, 1979, pp. 121-157.
3. M. Suwa, A.C. Scott, and E.H. Shortliffe, "An Approach to Verifying Completeness and Consistency in a Rule-Based System," *AI Magazine*, Vol. 3, No. 4, 1982, pp 16-21.
4. T.A. Nguyen et al., "Knowledge Base Verification," *AI Magazine*, Vol. 8, No. 2, Summer 1987, pp 69-75.
5. B. Cragun and H.J. Steudel, "A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems," *Int'l J. Man-Machine Studies*, Vol. 26, No. 5, 1987, pp. 633-648.
6. A.D. Preece, R. Shinghal, and A. Batarekh, "Principles and Practices in Verifying Rule-Based Systems," *Knowledge Eng. Review*, Vol. 7, No. 2, 1992, pp. 115-141.

rule. However, if the antecedents of one of these rules is swapped, the redundancy would not be certain, since the unified variables might have been instantiated to different values. On the other hand, consider

```
R4:  if type (?device, Iron) &
        temperature (?device, High)
     then problem (?device)

R7:  if type (?device, ?any_type) &
        temperature (?device, High)
     then problem (?device)
```

R4 and R7 might be redundant. The reason for uncertainty is that the value that will be provided for *?any_type* in the second rule might or might not be Iron. Therefore, UVT reports that R4 and R7 might be redundant.

Redundancies can cause serious problems. They might cause the same information to be counted several times, leading to erroneous increases in the certainty factors of their conclusions. Redundancies do not cause problems in systems where certainty factors are not involved and the first successful rule is the one to succeed. Redundant rules are not required to have the same certainty factors. In other words, rules with different certainty factors might still be redundant. This is also true for conflicting rules, subsumed rules, and rules with redundant If conditions.

Conflicting rules. Conflicting rules succeed in the same situation but produce conflicting results; that is, their antecedents are equivalent but their consequents conflict. For example, rules R4 and I3 definitely conflict with each other because they are unifiable and their conclusions are conflicting. On the other hand, consider

```
R1:  if type (?device, Iron) &
        temperature (?device, High)
     then works (?device)

R8:  if type (?device, ?any_type) &
        temperature (?device, High)
     then ~works (?device)
```

These rules might or might not conflict: They conflict only when the variable *?any_type* in R8 is instantiated to the value Iron.

Subsumed rules. If two rules' consequents are equivalent, and one rule's antecedent consists of the antecedents of the other and some additional literals, we say that the more restrictive rule (the one having more predicates in its antecedent) is subsumed by the other. When the more restrictive rule succeeds, the less restrictive one will also succeed. For example, consider

```
R8:  if type (?device, ?any_type) &
        temperature (?device, High)
     then ~works (?device)

R9:  if temperature (?device, High)
     then ~works (?device)
```

R8 is subsumed by R9 because R9 needs only a portion of information required by R8 to conclude that the device is not in working condition. In other words, when R8 succeeds, R9 also succeeds.

Redundant If conditions. Some rules contain unnecessary If conditions that can be removed without affecting the rule's effect. Such conditions make the inference engine do unnecessary work. Two types of unnecessary If conditions are possible. The first type occurs when a literal in one rule's antecedent conflicts with a literal in the other rule's antecedent and all the remaining literals in both the antecedents and the consequents of the rules are equivalent. For example, consider

```
R1:  if type (?device, Iron) &
        temperature (?device, High)
     then works (?device)
```

```
R10: if type (?device, Iron) &
        ~temperature (?device, High)
     then works (?device)
```

The predicate "temperature" in these rules is unnecessary because it cannot affect the conclusions of R1 and R10. These rules can be reduced semantically into a single rule:

```
RX : if type (?device, Iron)
     then works (?device)
```

The second type of redundant If condition occurs when two rules' consequents are equivalent, and one rule's antecedent contains a single literal that conflicts with a literal in the other rule's antecedent:

```
R1:  if type (?device, Iron) &
        temperature (?device, High)
     then works (?device)

R11: if ~temperature
        (?electronic_device, High)
     then works (?electronic_device)
```

We can combine these rules with a logical Or operation after unification through the substitution {*?device/?electronic_device*}:

```
RX:  if ~temperature (?device, High) Or
        temperature (?device, High) &
        type (?device, Iron)
     then works (?device)
```

Using the distribution property of Or over And operators, we can rewrite RX as

```
RX:  if (~temperature(?device, High) Or
        temperature (?device, High)) &
        (~temperature (?device, High) Or
        type (?device, Iron))
     then works (?device)
```

We can then simplify RX by removing the tautology:

```
RX:  if (~temperature (?device, High) Or
        type (?device, Iron))
     then works (?device)
```

Now, we can separate RX into two rules,

```
RX1: if type (?device, Iron)
     then works (?device)

RX2: if ~temperature (?device, High)
     then works (?device)
```

RX2 is equivalent to the original rule R11, whereas RX1 is the same as R1 except for the redundant If condition "temperature(?device, High)."

Dead-end rules. In backward-chaining systems, subgoals are created from the rules whose consequents match the current goal. Each subgoal must match a fact whose truth value is provided by the user, or the consequent of a knowledge base rule; otherwise, it is unreachable. A rule with unreachable antecedents is called a dead-end rule. We locate these rules by doing a completeness check on the knowledge base, that is, by identifying gaps in the knowledge base. We can then fix these gaps by adding the missing rules or marking the related facts as "askable." For instance, the rule

```
R12: if type (?device, Lamp) &
        lit (?device)
     then works (?device)
```

is a dead-end rule if there are no rules with consequent 'lit,' or if 'lit' is not set to be an askable predicate.

Cycles and contradictions within a rule. A cycle within a rule can be detected when the same predicate occurs in both its antecedent and consequent. For example, the predicate "hasPower" in R13 might cause a cycle:

```
R13: if connected (?point-x, ?point-y) &
        hasPower (?point-x)
     then hasPower (?point-y)
```

Contradictions occur when one literal conflicts with another in the same rule. This can happen in two ways. First, both conflicting literals can be in the rule's antecedents:

```
R14: if connected (?device-x, ?point) &
        connected (?device-y, ?point) &
        works (?device-x) &
        ~works (?device-y)
     then state (?device-y, Broken)
```

Here, the third and fourth literals might contradict each other, since the variables *?device-x* and *?device-y* might be instantiated to the same device.

The second type of contradiction in a rule occurs when one conflicting literal appears in the rule's antecedent and the other in its consequent. For example,

```
R15: if state (?device, OK) &
        ~works (?device) &
     then ~state (?device, OK)
```

R15's consequent and the first literal of its antecedent certainly contradict each other. However, some forward-chaining systems use this kind of rule to switch from one situation to another.

Dependencies between rules. Rules with common predicates in their antecedents and consequents might be naturally related, but these dependencies can lead to cycle problems. It is not theoretically hard to develop algorithms to check for cycles, but the existing algorithms in practice take too much time. There is no way to quickly check for cycles statically, so most verification and validation tools do this dynamically. Since our approach finds problems in the rule set before the expert system is used, UVT can only find *potential* independent cycles. Consider

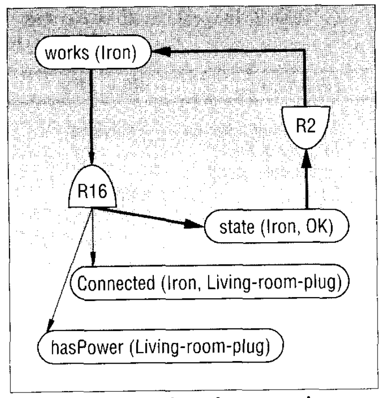


Figure 2. The dependency between rules causes a cycle (shown with heavier arrows).

```
R2:  if works (?device) &
     then state (?device, OK)

R16: if state (?device, OK) &
        connected (?device, ?point) &
        hasPower (?point)
     then works (?device)
```

To see how dependencies cause cycles, suppose that our goal is to find out if the steam iron works, that is, to try to satisfy "works(Iron)." A backward-chaining inference engine might choose R16 as relevant to this goal. Three subgoals are created from the antecedent of R16, as shown in Figure 2. The first subgoal is to determine the state of Iron. The first rule relevant to determining the state of a device is rule R2. Unfortunately, the antecedent of R2 requires that the steam iron works, which is the initial goal, and the inference engine enters into a cycle. This scenario is quite likely to occur in a large rule base, so detecting such cycles is a crucial job of a knowledge base verification tool.

UVT implementation

UVT is implemented in C and runs on Sun workstations. The tool starts by computing all the inferred rules and adding them to the rule base.

The next step is to find the relationships

Table 2. The contents of entries Rule_relation[5,6] and Rule_relation[1,8] in the Rule_relation table.

TABLE ENTRY	R5 VS. R6	R1 VS. R8
Consequent_relation	Equivalent	Conflicting
Equivalent_count	2	2
Conflicting_count	0	0
Type	Definite	May_be
Substitution list	{?appliance/?device, ?plug_out/?point}	{Iron/?any_type}

between rules and store them in two 2D tables. The Rule_relation[i,j] table contains the relational information obtained by comparing pairs of consequents and antecedents using unification. UVT compares the consequents first: They are either equivalent (unifiable), conflicting (negation of one is unifiable with the other), or different. If they are equivalent or conflicting, UVT stores the unification's substitution list. Using this same list, UVT then compares the antecedents and finds the equivalent, conflicting, and different literals and whether each relationship is definite or not. UVT identifies each possible relationship between rules as definite if every variable in the substitution list is unified with another variable, and if all variable pairs are instantiated in the predicates in which they are unified. In this process, the substitution list might be augmented, since there might be variables in the antecedents' literals that do not occur in the consequents but that can unify.

Each entry in the Rule-relation[i,j] table contains the following information:

- consequent_relation: Equivalent, Conflicting, or Different
- equivalent_count: the number of equivalent literals in the antecedents
- conflicting_count: the number of conflicting literals in the antecedents
- type: whether the possible relationship is Definite or May_be
- substitution list: the substitutions used in the unification

Table 2 shows the entries of the Rule_relation table corresponding to the rule pairs R5/R6 and R1/R8.

UVT examines each entry of the Rule_relation table to detect redundant, conflicting, and subsumed rules, and rules having redundant If conditions. If it identifies a problem between rules R_i and R_j, UVT issues a warning message if Rule_relation[i,j].type = May_be, or an error message otherwise.

The following rule determines if rules R_i and R_j are redundant:

```
if Rule_relation[i,j].consequent_relation
      = Equivalent &
   |Ante(Ri)| = |Ante(Rj)| =
      Rule_relation[i,j].equivalent_count
then Ri and Rj are redundant
```

where $|Ante(R_i)|$ represents the number of literals in R_i's antecedent. That is, if the rules' consequents are equivalent and the number of literals in the rules' antecedents equals the number of literals found to be equivalent, then R_i and R_j are redundant.

UVT identifies R_i and R_j as conflicting and subsumed using these rules:

```
if Rule_relation[i,j].consequent_relation
      = Conflicting &
   |Ante(Ri)| = |Ante(Rj)| =
      Rule_relation[i,j].equivalent_count
then Ri and Rj are conflicting

if Rule_relation[i,j].consequent_relation
      = Equivalent and
   |Ante(Ri)| > |Ante(Rj)| &
   |Ante(Rj)| =
      Rule_relation[i,j].equivalent_count
then Ri is subsumed by Rj
```

UVT identifies rules R_i and R_j as having redundant If conditions using these rules:

```
First type of redundancy:
if Rule_relation[i,j].consequent_relation
      = Equivalent &
   |Ante(Ri)| = |Ante(Rj)| &
   Rule_relation[i,j].conflicting_count =
      1 &
   |Ante(Ri)| =
      Rule_relation[i,j].equivalent_count+1
then Ri and Rj have redundant If
   conditions

Second type of redundancy:
if Rule_relation[i,j].consequent_relation
      = Equivalent &
   (|Ante(Ri)| = 1 &
   |Ante(Rj)| > 1 or |Ante(Rj)| = 1 &
   |Ante(Ri)| > 1) &
   Rule_relation[i,j].conflicting_count = 1
then Ri and Rj have redundant If
   conditions
```

To detect possible problems within the rules, UVT also examines each rule individually. To detect cycles in a rule, UVT compares each literal in the antecedent to the literal in the consequent. If they are the same, a cycle is reported. To detect contradiction in a rule, UVT compares the antecedent's literals to each other and to the literal in the consequent. If the compared literals are conflicting, a contradiction is reported. To detect a dead-end rule, UVT checks all the literals in the rule's antecedent to see whether they match some fact or rule. If at least one literal does not have a match, the rule is identified as dead-end.

The second table in UVT, called If_then[i,j], is used to detect potential cycles due to the dependencies between rules. Each entry stores information about whether R_j's consequent occurred in R_i's antecedent. For example, if the entry If_then[2,16] has the value Equivalent, this indicates that one of the antecedents of R2 is the same as the consequent of R16. Similarly, If_then[16,2] has the value Equivalent, as well. On the other hand, if If_then[1,4] has the value Different, none of the antecedents of R1 is the same as the consequent of R4. Using this table, UVT reports possible cycles by listing the sequences of rules involved in the cycles.

UVT spends most of its time finding inferred rules. The larger the number of interrelated rules (with common predicates in their antecedents and consequents), the longer it takes to find inferred rules. As the program builds the relationship tables, it compares each rule with the others. The time spent on building the tables and detecting the problems have complexity $O(M^2)$, where M is the number of rules (N) plus the number of inferred rules.

To reach conclusions, UVT takes into account the dependencies between literals in a rule, the relationships between rules, and the way unification occurs. It then generates two types of messages in the form of warnings and errors, using the word "may" when it cannot identify definite problems.

A COMMON PRACTICE IN BUILDING knowledge-based systems is to avoid potential conflicting situations through analysis and consistency checking of the knowledge base during development. This is costly, especially as the amount and diversity of knowledge increases. Gathering diverse knowledge from different sources, resolving knowledge incompatibility problems, and dividing domain knowledge into smaller, internally consistent collections are all difficult problems.

Recently, autonomous distributed knowledge-based systems have gained much attention. In domains such as distributed sensing, medical diagnosis, and air traffic control, knowledge is inherently spatially distributed. Development-time verification approaches can be used to detect and resolve inconsistencies and incompleteness in individual KBSs, but inconsistencies between systems must also be resolved. We can avoid these problems by allowing participating systems to generate conflicting solutions to subproblems and then resolving them at runtime. Strategies such as backtracking, compromise negotiation, integrative negotiation, constraint relaxation, case-based and utility reasoning methods, and multiagent truth maintenance[6-10] can be used to resolve conflicts. Most systems for runtime conflict resolution use either centralized approaches or bilateral negotiation techniques. A new approach could be to develop a negotiation paradigm that allows multiple systems to reconcile their differences and resolve inconsistencies.

References

1. T.J. O'Leary et al., "Validating Expert Systems," *IEEE Expert*, Vol. 5, No. 3, June 1990, pp. 51-58.

2. M. Suwa, A.C. Scott, and E.H. Shortliffe, "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System," *AI Magazine*, Vol. 3, No. 4, 1982, pp 16-21.

3. F. Polat and H.A. Guvenir, "Knowledge Base Verification in an Expert System Shell," *Proc. Fourth Int'l Symp. on Computer and Information Sciences*, Vol. 2, METU, Ankara, Turkey, 1989, pp. 889-898.

4. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing, Palo Alto, Calif., 1980.

5. J.H. Siekmann, "An Introduction to Unification Theory," in *Formal Techniques in Artificial Intelligence: A Source Book*, R.B. Banerji, ed., Elsevier North-Holland, Amsterdam, 1990, pp 369-409.

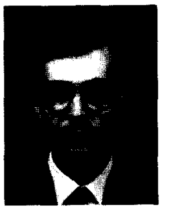
6. M.N. Huhn and D.M. Bridgeland, "Multiagent Truth Maintenance," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 21, No. 6, Nov./Dec. 1991, pp. 1,437-1,445.

7. M. Klein and S.C-Y. Lu, "Conflict Resolution in Cooperative Design," *Artificial Intelligence in Engineering*, Vol. 4, No. 4, 1989, pp. 168-180.

8. S. Lander and V.R. Lesser, "Conflict Resolution Strategies for Cooperating Expert Agents," *Int'l Conf. on Cooperating Knowledge-Based Systems*, Springer-Verlag, New York, 1990, pp. 183-198.

9. F. Polat and H.A. Guvenir, "A Conflict-Resolution-Based Cooperative Distributed Problem-Solving Model," *Proc. AAAI '92 Workshop on Cooperation among Heterogeneous Intelligent Agents*, AAAI, Menlo Park, Calif., 1992, pp. 106-115.

10. K.P. Sycara, "Negotiation Planning: An AI Approach," *European J. of Operational Research*, Vol. 46, No. 2, 1990, pp. 216-234.



Faruk Polat is a doctoral candidate in computer engineering and information science at Bilkent University in Ankara, Turkey. He has been working on his dissertation as a visiting NATO science scholar at the University of Minnesota, Minneapolis, for the 1992-1993 academic year. His research interests include knowledge-based systems, knowledge base verification, and distributed artificial intelligence. He received his BS in computer engineering from the Middle East Technical University, Ankara, in 1987 and his MS degree in computer engineering and information science from Bilkent University in 1989.

H. Altay Guvenir is an assistant professor of computer engineering and information science at Bilkent University. His research interests include expert systems, machine learning, and computational linguistics. He received his MS in electrical engineering from Istanbul Technical University in 1981 and his PhD in computer science from Case Western Reserve University in 1987. He is a member of AAAI, ACM, ACM SIGArt, and the International Association of Knowledge Engineers.

Readers can reach the authors at Bilkent University, Dept. of Computer Engineering and Information Sciences, 06533 Bilkent, Ankara, Turkey; e-mail, polat@trbilun.bitnet or guvenir@trbilun.bitnet