BY YUE KANG AND A. TERRY BAHILL

a TOOL *for* EXPERT-SYSTEM

Test cases can find run-time errors in the system every time a rule is added or modified

The most difficult part of expert-system design is testing. Brute-force enumeration of all inputs is impossible for most systems, so the traditional testing method has a human expert run many test cases on the expert system. This method is time-consuming and fallible. Furthermore, the knowledge engineer never knows when enough test cases have been run. Our run-time tool helps the knowledge engineer know exactly when to quit.

More mistakes could be corrected if many experts tested the system. It is often possible to find an expert who will devote lots of time for interviewing, debugging the knowledge base, and running test cases. It is much harder to find other experts with the time to test the final product: their time is expensive; there are few of them in any geographical area; and they don't have the personal commitment to the project. Because experts have so many time constraints, there should be tools to help make judicious use of it.

Evaluating an expert system by using test cases is certainly not original. P.G. Politakis[1] has shown how statistics gathered while running test cases can be used by the developer to modify the rules and improve the expert system.

The technique described here comes into play at run-time. Each rule-firing is recorded. Rules that never succeed and rules that succeed for all test cases are probably mistakes, and the human expert is notified. "Succeed" means all the premises are true and the expression in the conclusion is assigned the appropriate value.

RUNNING TEST CASES

It appears we are back to square one because we need to bring in the expert to run test cases. However, only one test case is necessary; the expert does not have to reanswer 100 questions every time the knowledge engineer corrects a spelling error. To create test cases, start the system and ask the expert to pull the file or visualize a particular patient. The expert should run a consultation and answer all the questions. At the end of the consultation, after the expert system gives its advice (but before you exit or restart), save the intermedi-

DETECTING ERRORS

ate knowledge. (For M.1 the command is *savecache.*) Eliminate the values derived by the inference engine with a text editor, leaving only the human answers to questions. (M.1 tags these "because you said so." Unfortunately, this tag doesn't work if the expert answers "unknown" to a question; the knowledge engineer still has some work to do.)

To create a second test case, ask your expert to concentrate on another patient (or malfunctioning circuit, for example) and produce another set of answers. Save this set under a different name. Convenient names may be *case1, case2, case3,* and so on. In this manner the expert can produce many test cases that can be used for subsequent testing during the project's lifecycle. Every time a rule is added or modified these test cases can be automatically played through the system to look for run-time errors. If you did not add new questions to the knowledge base, the advice should be produced immediately. If you added new questions, they will be asked, and you'll get your advice and can save the new test case. If the modifications require more knowledge from the expert, it may be possible to glean this information from a telephone call. The test cases could be readily updated without demanding a lot of the experts time and you could test the system at your leisure.

Using real test cases provided by the expert is only one method of exercising a knowledge base. Test cases may also be invented by the knowledge engineer and provided by the user; those created by the knowledge engineer are often more efficient, because more rules can be exercised by fewer test cases. The expert, the knowledge engineer, and the user are like three blind men describing an elephant: each understands a different aspect of the system, so their test cases will exercise different portions of the knowledge base. It may be wise to use test cases from all three sources.

Rules never succeed if either of two classes of mistakes are made: failure due to false premises and failure due to single-value cutoff. The problems and techniques discussed are universal and should apply to any rule-based expert system (we've tried them on M.1, VP-Expert and Personal Consultant Plus systems).

Validator

Validator is a program that interactively verifies and validates rule-based expert-system knowledge bases. It checks the syntax and semantics for potential errors and brings them to the attention of the knowledge engineer. Validator does not attempt to fix errors; that task is left to the knowledge engineer. The system has six modules: a preprocessor, syntax analyzer, syntactic error checker, debugger, chaining thread tracer, and knowledge-base completeness module.

Validator was tested on 67 expert systems selected from MA projects, potential commercial systems, class projects for graduate courses in knowledge engineering, and commercial demonstration systems. The potential mistakes flagged by Validator fell into nine categories: illegal use of reserved words; rules that could never fire (both backward and forward rules); unused facts; unused questions; unused legal values; repeated questions; multiple methods (including expressions that appear in questions and facts, questions and conclusions, and facts and conclusions); rules using illegal values (including mismatches between any of the sets of legal values, utilized values, concluded values, and assigned values); and incorrect instantiations. Validator was written and is maintained by Musa Jafar and Terry Bahill, at BICS, 1622 W. Montenegro, Tucson, Ariz. 85704.

FAILURE DUE TO FALSE PREMISES

Consider this type of *if-then* production rule: *if premise = yes then conclusion = yes.* If the premise cannot be satisfied in any consultation, then the rule will fail. For example, in *if a>1 then c = 0,* if the designer thought that *a* would sometimes be larger than one (but in the real world *a* was always less than one), then the premise could never be satisfied and the rule would never succeed.

If a premise has many conditions, they may contradict each other, so the premise is always false (such as *if humidity = 3% and heavy-rain = yes then conclusion.)* The contradictory conditions could even be indirect and spread out in other rules. For example:

```
    goal = c.
rule-1: if heavy-rain = yes and b = yes then c = yes.
rule-2: if humidity = 3% then b = yes.
```

rule-1 cannot succeed. We can generalize this idea to:

```
rule-1: if heavy-rain = yes and b1 = yes then c = yes.
rule-2: if ... b2 = yes and ... then b1 = yes.
.   .   .   .
.   .   .   .
.   .   .   .
rule-k: if ... bk = yes and ... then bk-1 = yes.
rule-k+1: if humidity = 3% then bk = yes.
```

Since *b1* depends on *bk, rule-1* can't succeed.

If a rule has one premise that will be false for all valid combinations of values, the rule will never succeed and is probably a mistake. Notice that this type of error-checking depends on knowledge and therefore can be used only at run-time.

FAILURE DUE TO CUTOFF

When seeking a value for a single-valued expression, most backward-chaining expert systems stop after finding a value with complete certainty. If a system always stops before a certain rule, that rule will never succeed. Consider this knowledge base:

```
    goal = c.
rule-1: if a1 = yes then c = 1 cf 100.
.   .   .   .
.   .   .   .
.   .   .   .
rule-k: if ak = yes then c = k cf 100.
rule-k+1: if b = yes then c = k+1 cf X.
```

Assume *c* is single-valued. If one of *a1, a2, ... ak* is always 100% yes, then the system will always stop seeking a value for *c* before *rule-k+1* can succeed. Therefore, *rule-k+1* will never succeed. Consider this set of rules:

```
rule-1: if a = yes then c = 1 cf 100.
rule-2: if a = no then c = 2 cf 100.
rule-3: if b = yes then c = 3 cf X.
```

rule-3 will never succeed. After the inference engine has found a value with 100% certainty, it won't seek further values. (This example presumes the user is not allowed to answer "unknown" when a value for *a* is queried.) These errors cannot be found by syntax checking because they are semantic; they can be found only during run-time investigations. Validator (see sidebar) scans the knowledge base by flagging single-valued terms with different values attributed to them in different rules as potential errors.

In addition to these types of errors, our run-time tool detected rules that could never succeed due to other types of errors. Validator had already detected many of these mistakes, but it produces some false positives, so knowledge engineers sometimes ignore the warnings.

Rules that succeed for every test case are probably mistakes. If a rule is always true then it might be simpler to replace it with a fact. Of course, some control rules always succeed, and some rules will be designed for rare situations not exercised by the test cases at hand. Again, this technique is only advisory; the expert must make the final decision about the rule's correctness.

To detect which rules succeed at runtime, we add additional terms to the knowledge base. For example, this original knowledge base:

```
rule-1: if premise-1 = yes then conclusion-1.
.    .    .    .
.    .    .    .
.    .    .    .
rule-k: if premise-k = yes then conclusion-k.
```

must be edited to become this transformed knowledge base:

```
rule-1: if premise-1 = yes
   then conclusion-1 and r-1 = yes.
.    .    .    .
.    .    .    .
.    .    .    .
rule-k: if premise-k = yes
   then conclusion-k and r-k = yes.
```

We use the terms *r-i* to record which rules succeeded (*i* is an index that goes from one to *k,* where *k* is the number of rules in the knowledge base). When the system invokes a rule, if the premises are true (meaning the rule succeeded), *r-i* is assigned a value of *yes.* If rule *i* has succeeded during the consultation, the value of *r-i* will be *yes.* If rule *i* has not succeeded during the consultation, the value of *r-i* will not be *yes.*

This technique works for both forward- and backward-chaining shells, but it does require the shell to allow multiple terms in the rule conclusions. Personal Consultant Plus from Texas Instruments, VP-Expert from Paperback Software, and many other shells allow this technique. However, to make the technique work for M.1, an additional clause must be added as the last premises of each rule, as shown in this modified knowledge base:

```
rule-1: if premise-1 = yes and do(set r-1 = yes)
   then conclusion-1.
.    .    .    .
.    .    .    .
.    .    .    .
rule-k: if premise-k = yes and do set(r-k = yes)
   then conclusion-k.
```

This example is written as if there were only one premise, but the technique will work with multiple premises as long as the *do set(r-k = yes)* is the last one. When *rule-i* fires, if the original premises are *true* (meaning the rule will succeed), then *r-i* will be set to *yes.* If the premises are *false,* then the rule will fail and *r-i* will not be set to *yes.*

None of these additions influences the function of the rules, but they do allow detection of which rules succeeded. The next step is storing the values of the *r-i*, which reflect the status of a rule in only one consultation. Since our goal is to determine which rules have succeeded after many consultations, we must accumulate the values of *r-i* from the first to the last consultation. From the results of this accumulation we can find the rules that never succeeded and that succeeded for every test case.

RECORDING SUCCESSFUL RULES

We wrote a C program to modify the knowledge bases automatically. It added *and do set(r-i = yes)* into rules *kb-1* to *kb-k* and appended *kb-k+1* to *kb-k+3* to the original knowledge base to record which rules succeeded. Examine this modified M.1 knowledge base:

```
    goal = goal-1.
kb-1: if premise-1 = yes and do set(r-1 = yes)
    then conclusion-1.
.    .    .    .
.    .    .    .
.    .    .    .
kb-k: if premise-k = yes and do set(r-k = yes)
    then conclusion-k.
kb-k+1: goal = goal-2.
kb-k+2: if do(log filename1) and do(show r-X) and
    do(log off) and external(cprogram1,[]) = [] and
    printrule = yes and do(log filename2) and
   do(uses do(set r-X=yes)) and do(log off) and
    external(cprogram2,[]) = [] then goal-2=true.
kb-k+3: question(printrule) = 'Would you like to
    create a file showing the rules and the number of
    times they succeeded?'
```

To use our program we ran a consultation with the modified knowledge base. We saved test cases, loaded one with the *loadcache* command, and restarted. The original goal, *goal-1,* drove a normal consultation,

TABLE 1. Summary of run-time tool testing results.

NAME	SIZE (KB)	KB ENTRIES	RULES	UNSUCCESSFUL RULES	TEST CASES
TV-FILM	32	290	25	0	6
BACKUP	8	50	20	1	8
AUTOMECH	8	56	24	0	12
MACEXPERT	11	85	16	1	15
SOFTWARE	7	68	32	1	26
ADVISOR	84	307	64	2	30
HELPER	63	204	57	3	40
DIET	14	118	33	11	8
WIREBOND	19	113	39	22	6
DRUGDEAL	26	206	81	34	7
VOLCANIC	18	141	65	36	8
DRUG	40	120	82	36	13
CHROMIE	114	681	231	41	20
VET	33	235	105	67	6
DELL	54	453	148	70	6
CARPET	56	390	106	70	20

```
                              Example 1.
goal=intro.
rule-1: if a_banner_displayed is sought
   and output is sought
   and class_assignment is sought
   and do(reset done)
   then intro.
rule-2: if not(class)
   then intro.
/*Because intro was single-valued and rule-1 always succeeded, rule-2 never
 fired. Earlier we called this failure due to single valued cutoff.*/

                              Example 2.
goal = complete-commands.
rule-1: if help-with-commands=yes
   and commands is sought
   then complete-commands.
rule-2: if help-with-commands=no
   then commands.
/*This is a special case of failure due to false premises. It could be
simplified to goal=c.*/
rule-1a: if a=yes
   and b is sought
   then c=yes.
rule-2a: if a=no
   then b=yes.
/*Since a cannot be yes and no at same time, rule-2a can never succeed.*/

                         Example 3 (simplified).
goal=c.
rule-1: if a is unknown
   and b=yes
   then c=yes.
rule-2: if b=W
   and d=V
   and computer(W,V)=X
   then a=X.
fact-1: computer(yes,V)="something".
/*If b=no rule-1 fails. If b=yes, then a=something and is therefore known.
 Therefore, rule-1 can never succeed. This is another example of failure due
 to false premises.*/

                         Example 4 (simplified).
goal=c.
rule-1: if a=1
   then c=yes.
rule-2: if not(b=1)
   and b=X
   then a=X.
rule-3: if b=1
   and d=X
   then a=X.
legalvals(d)=[2,3].
/*Neither rule-2 nor rule-3 could make a=1, so rule-1 will never succeed. Using
variables makes it harder to see the contradiction.*/
```

LISTING 1. Examples.

and *goal-2* caused M.1 to invoke rule *kb-k+2,* which created a results file called *filename1.* This file contained the cache entries for all successful rules. For example, if *rule-5* succeeded then *r-5 = yes because...* would be in cache and put into *filename1.*

Next, we ran a another consultation (loaded another test cache) and our program added the new statistics to the file. The external function *cprogram1* consolidated the new data with the data from previous consultations. When we finished running all the test cases, we answered "yes" to the question "Would you like to create a file showing the rules and the number of times they succeeded?" Rule *kb-k+2* then caused M.1 to create a file called *filename2.* The external function *cprogram2* modified that file, producing the final output file containing the rules, the knowledge-base number of each rule, and the number of times each rule succeeded. (In a consultation, the new knowledge base works the same as the old one. The user sees no differences.)

We tested this run-time tool on 16 expert systems. Table 1 lists their names, sizes, number of knowledge-base entries, rules, rules that never succeeded during our tests, and test cases we used to exercise the expert systems. We identified each rule that never succeeded and determined why not. Many rules failed because they contained mistakes. The four examples shown in Listing 1 summarize these mistakes.

Our most powerful verification and validation tool, Validator,[2,3] could not detect these mistakes because they depend on the user's answers and can be detected only at run-time.

The first seven expert systems listed in Table 1 had fewer mistakes, and we could determine the cause of failure for every rule that never succeeded. These seven expert systems had an average of 34 rules, ran an average of 20 test cases for each system, and had an average of one unsuccessful rule per system.

For the next nine systems listed in Table 1, we could not find the cause of failure for every unsuccessful rule. These nine systems had an average of 99 rules, they were given an average of 10 test cases and had an average of 43 unsuccessful rules. So for the systems we tested, as the number of test cases approached one-half the number of rules, the number of unsuccessful rules diminished.

The best way to test an expert system is to let the domain expert run consultations, first creating a test case for every 10 rules (10 test cases for a 100-rule system) and observing which rules never succeed. Next the expert should provide test cases that might make these rules succeed. This procedure could be repeated until most of the rules have succeeded. Then the knowledge engineer could try to find the reason for the remaining rules not succeeding.

LIMITATIONS

This technique is limited in one way because a rule that never succeeds is not necessarily wrong; it is possible that the test cases just did not exercise that rule. For example, for Chromie (Table 1) the expert provided many test cases, yet 41 rules never succeeded. When this statistic was revealed to the expert, she said, "Of course. That part of the knowledge base is so simple, I never bothered to give you a test case that would exercise it."

Even if a rule succeeds we cannot be sure that it contributes to the final result:

```
rule-1: if a = yes then c = 1 cf 70.
rule-2: if b1 = yes then c = 2 cf 100.
.    .    .    .
.    .    .    .
.    .    .    .
rule-k: if bk = yes then c = k cf 100.
```

Assume *c* is single valued. If one of *b1* to *bk* is always *yes,* even if *rule*-1 succeeded, $c = 1$ *cf 70* would be replaced by a result from another rule. So *rule-1* is useless.

TECHNIQUE BENEFITS

Testing expert systems is difficult, time consuming, and unfulfilling. We have developed a technique to make this task easier and to give the designer more confidence in the final product. This technique is intended to be run after all static errors have been removed from the knowledge base. For example, spelling errors, use of reserved words, and rules that do not link to the rest of the knowledge base can be detected in a careful reading of the knowledge base or with an interactive tool such as Validator. After this detection is complete, our run-time tool can reveal more potential errors using knowledge stored from consultations with the expert. (It detects rules that never succeed and rules that succeed for every test case.) Such rules are probably mistakes and should be brought to the attention of the knowledge engineer.

We have shown two classes of mistakes that would cause a rule never to succeed: the premises could always be false, or single-valued cutoff might cause the inference engine to stop seeking a value for a term before a certain rule is ever encountered. AI

REFERENCES

1. Politakis, P.G. *Empirical Analysis for Expert Systems.* Boston, Mass.: Pitman Advanced Publishing Program, 1985.

2. Jafar, M.J., and A.T. Bahill. "Validator, a tool for verifying and validating personal computer based expert systems." In Brown, D.E., and C.C. White (eds.) *Operations Research and Artificial Intelligence: The Integration of Problem Solving Strategies.* Boston, Mass.: Kluwer Academic Publishers, 1990.

3. Jafar, M.J. "Tool for Interactive Verification and Validation of Rule Based Expert Systems." Ph.D. dissertation, Dept. of Systems and Industrial Engineering, University of Arizona, Tucson, 1989.

Yue Kang is a graduate student systems and industrial engineering at the University of Arizona, Tucson, and was previously a computer engineer in Beijing. A. Terry Bahill is a professor of systems engineering at the University of Arizona and is an Associate Editor of *IEEE Expert.*

GA TECH Expert Systems

Short Courses

leading to a

CERTIFICATE IN KNOWLEDGE ENGINEERING

Let Georgia Tech knowledge engineers show you how to lead your company to the forefront by employing expert systems on personal computers.

This complete series is offered in spring 1990:

1. *K.E. I -- Manager's View of Expert Systems*
   March 19
2. *Knowledge Acquisition*
   March 20-21
3. *Lisp-at-Work: Essential Syntax and Use*
   March 22
4. *K.E. II -- Rapid Prototyping Practicum*
   March 23-24
5. *Object-Oriented Programming Systems*
   March 28-29
6. *K.E. III -- Project Management*
   March 30

For more information and course registration call:

Education Extension—M
Georgia Institute of Technology
Atlanta, Georgia 30332-0385
1-800-325-5007

CIRCLE 19 ON READER SERVICE CARD