Interactive Verification of Knowledge-Based Systems

Musa Jafar, Washington State University
A. Terry Bahill, University of Arizona

THE VALIDATOR PROGRAM INTERACTIVELY CHECKS THE CONSISTENCY AND COMPLETENESS OF A KNOWLEDGE BASE. IT SYSTEMATICALLY POINTS OUT POTENTIAL ERRORS TO THE KNOWLEDGE ENGINEER TO HELP VERIFY AND VALIDATE A RULE-BASED EXPERT SYSTEM.

THE TYPICAL DEVELOPMENT cycle for a knowledge-based system consists of

- Verification, or building the system right: This ensures that the system correctly implements the specifications and determines how well each prototype conforms to the design requirements. Verification guarantees product consistency at the end of each phase (with itself and with the previous prototypes), and ensures a smooth transition from one prototype to another.
- Validation, or building the right system: This ensures the consistency and completeness of the whole system.[1]
- Developmental testing: Running test cases to explore the system and expose errors.
- Final testing or evaluation: Running as many test cases as possible and watching the system's input-output behavior to evaluate its performance and determine its usefulness.

Verification, validation, and developmental testing are carried out during and at the end of each prototype phase. For expert systems, verification traditionally involves three steps:

(1) The knowledge engineer looks for syntactic or semantic errors in the knowledge base
(2) The knowledge engineer runs test cases to find mismatches between the system's solutions and the expert's
(3) The expert runs test cases on the system.

This process takes a lot of time and still doesn't guarantee an error-free knowledge base, since for most systems it's impossible to enumerate all inputs. Even when an expert system has been fully verified and validated, its acceptance by the customer is not guaranteed. Final testing and evaluation still must be done after development and before deployment. This is a complicated process that can include statistical hypotheses testing and building of confidence intervals.

Researchers have raised several ideas and heuristics for traditional verification (such as walkthroughs and syntax checking),[2,3] as well as work on the consistency and completeness of logic-based systems.[4,5] Most tools for verifying, validating, and acquiring knowledge for knowledge-based systems came from big projects designed mainly for automatic knowledge acquisition: Teiresias, Emycin, Mole, Seek and Seek2,[6] AutoIntelligence,[7] Expertise Transfer System, Acquinas, and so on. The main objective of these systems is to communicate directly with the expert, using formal interviews coupled with protocol analysis (example problems), classification interviews (repertory grids), and concepts from personal-construct theory to help the knowledge engineer build a domain knowledge base. The power and capabilities of these

0885/9000/93/0200-0025 $3.00 © 1993 IEEE

```
Delete(Course, Number) = Deleted
Delete(Course, Number) = Ignored
.
.
.
Demester(Sem) = Yes
.
.
.
Semester(Sem) = Checked
Semester(Sem) = Ignored
Semester(Sem) = Yes
```

(a)

```
Help-With = Commands
Help-With = Basic
Help-With = Editor
Help-With = Yes
.
.
.
Restart(System) = 1
Restart(System) = 2
Restart(System) = 3
Restart(System) = 4
Restart(System) = Yes
```

(b)

Figure 1. Two syntax errors discovered in existing knowledge bases: A misspelled word (a), and an incorrect value (b).

tools is limited: They are good only for classification-type problems.

Another group of tools keep the knowledge engineer in charge of the knowledge transfer process: They include Oncocin, Cover,[8] the ART Rule Checker (ARC),[9] the Expert-System Validation Associate (EVA)[10] and the Expert-System Checker (ESC).[11] Such tools let the knowledge engineer build complete, high-quality knowledge-based systems, debug the knowledge-base syntax, and check its semantics.

A third group of tools includes Xcon, Mycin, Internist, and Prospector.[12] Simulation validation techniques have also been suggested, but they are domain-dependent and their effectiveness in building statistical models for hypothesis testing is limited unless the knowledge engineer can come up with quantitative factors to compare the system's performance with an expert's.[13]

Our Validator program fits into the second group, although it is broader and offers an interactive design. Validator verifies and validates rule-based expert systems, and guarantees that every element in the knowledge base is accessible and essential to the system. The program checks for syntactic errors, unused rules, facts, and questions, incorrectly used legal values, redundant constructs, rules that use illegal values, wrong instantiations, and multiple methods for obtaining values for expressions. Most of its features are generic and founded in software verification and logic systems.

Verification with Validator

Validator scans a knowledge base and checks its syntax and semantics for possible errors, which it brings to the knowledge engineer's attention. The program leaves the task of fixing such errors to the engineer. The verification part of Validator has four modules: a preprocessor, a syntax analyzer, a syntactic error checker, and a debugger.

After the knowledge engineer runs the knowledge base through a spelling checker, Validator's preprocessor performs a low-level syntax check on it. Syntactic errors can cause the production language (which here refers to AI languages as well as expert-system shells) to misinterpret a knowledge base and consequently alter its syntactic structure, leading to semantic errors. Detecting such errors saves the knowledge engineer and the expert much time, frustration, and grief. This type of checking is dependant on the production language, and ameliorates the shortcomings of the language's compiler.

Validator then builds internal representation structures of the knowledge base,[12] which the other three modules analyze for syntactic compliance. These techniques do not decrease the importance of developmental testing; rather, they speed system development and give the knowledge engineer and experts more time to concentrate on areas that need human attention.

The syntax analyzer. Many syntactic errors are caused by misspellings, typographical errors, or ill-formed knowledge-base structures.[8,9] In fact, many expert systems we tested contained misspelled words and illegal values, ranging from simple mistakes with no effect on the knowledge base to major problems that altered the whole system structure. For example, the list of all possible conclusions in one system in Figure 1a contains a misspelled word ("demester"), while the list of premises in another system in Figure 1b shows an incorrect value ("yes") given to the expressions Help-With and Restart.

By manually performing an "objective, step-by-step review" of the knowledge base,[1] an experienced knowledge engineer can easily detect such problems if the expressions are displayed close to each other and are directly connected. The syntax analyzer therefore takes the internal representation structures and creates a knowledge base dictionary: an alphabetical listing of the expressions in the knowledge base and their categories: goals, rule premises, rule conclusions, questions (with legal values), and facts (with values). Especially helpful is the fact that a knowledge base dictionary can be read in sections more easily than can an entire knowledge base. For example, a developer forced to read an entire knowledge base would be hard pressed to remember if a hyphen or an underscore was used to tie two words together.

Figure 2 shows the knowledge base dictionary derived from applying Validator to a section of the Poison Control Expert System. The "†" indicates a value where an error was later flagged. Validator didn't flag "comma" — a value for Symptom (Aspirin) — as an error because the person who built the knowledge base misspelled the word the same way every time (it should be "coma"). In contrast, Validator caught the inconsistent spelling of "aspirin" ("aspirine").

The syntactic error checker. People can easily detect inconsistent expressions if presented in pairs rather than in large collections, but machines are better at detecting global and indirect inconsistencies, such as those that can occur when a knowledge engineer lacks knowledge about the production language or the application domain. The syntactic error checker therefore looks for syntax that, although legal, produces unspecified behavior by the production language. Such "out-of-range values" (such as incorrect usage of reserved words) usually escape detection by the production language and the knowledge engineer. Validator detects these errors in the context in which they occur.

Out-of-range values. Expressions get their values from facts, from user responses to questions, or from the conclusions of rules. There are several kinds of values:

```
KNOWLEDGE BASE DICTIONARY:
*List of goals:
Begin
Finished
Treatment

*Expressions and their utilized values:
Dose(Bill, Drug) = A
Number(Bills) = Bills
Quantity = [High, Low, Mid]
Symptom(Acetaminophen) = [Anorexia, Anrexia†, Vomiting,
Pallor, Yes†]
Symptom(Acetylsalicylic-Acid) = [Vomiting, Yes†]
Symptom(Aspirin) = [Blood, Comma, Confusion, Delirium,
Dizziness-And-Ear-Tingling, Dizziness-And-Ear-Tingling†,
Elevation-Of-Temp, Increase-Sweating, Psychosis, Stupor,
Tinnitus, Vomiting, Yes†]
Symptom(Aspirin-And-Caffeine) = [Nausea, Yes†]
Symptom(Paracetamol) = [Nausea, Vomiting, Yes†]
Type(Drug) = [Aspirin, Aspirine†, Acetaminophen, Acetylsalicylic-
Acid, Aspirin-And-Caffeine, Paracetamol]
Victim = [Adult, Infant, Kid]

*Expressions and their concluded values:
Begin = [Yes, No]
Finished = [No]
Quantity = [Low, Mid, High]
Type(Drug) = [Acetaminophen, Acetylsalicylic-Acid, Aspirin,
Aspirin-And-Caffeine, Paracetamol]
Treatment = [Yes]
Victim = [Adult, Infant, Kid]

*Question expressions and their legal values:
Age(Victim) = Number(0, 120)
Continue-Search = [Yes, No]
Dose(Bill, Drug) = Number(0, 1)
Quantity(Bill) = Number(0, 20)
Start = [Yes, No]

Symptom(Acetaminophen) = [Pallor, Nausea, Vomiting, Anorexia]
Symptom(Acetylsalicylic-Acid) = [Elevation-Of-Temp, Excitability,
Headache, Insomnia, Irritability, Increase-Heart-Rate, Muscle-
Tremor, Nausea, Vomiting Increase-Sweating]
Symptom(Aspirin) = [Blood, Comma, Confusion, Delirium,
Dizziness-And-Ear-Tingling, Elevation-Of-Temp, Increase-
Sweating, Nausea, Psychosis, Stupor, Tinnitus, Vomiting]
Symptom(Aspirin-And-Caffeine) = [Excitability, Headache,
Increase-Heart-Rate, Insomnia, Irritability, Muscle-Tremor,
Nausea]
Symptom(Paracetamol) = [Pallor, Nausea, Vomiting, Anorexia]
Type(Drug) = [Acetaminophen, Acetylsalicylic-Acid, Aspirin,
Aspirin-And-Caffeine, Paracetamol]

POSSIBLE MISTAKES:
*Out-of-range values and their expressions:
Symptom(Acetaminophen) = Yes
Symptom(Acetaminophen) = Anrexia
Symptom(Acetylsalicylic-Acid) = Yes
Symptom(Aspirin) = Yes
Symptom(Aspirin) = Dizziness-And-Ear-Tingling
Symptom(Aspirin-And-Caffeine) = Yes
Symptom(Paracetamol) = Yes
Type(Drug) = Asprine

*Never used values and their expressions:
Continue-Search = [No]
Symptom(Acetaminophen-Or-Paracetamol) = [Anorexia, Nausea,
Pallor]
Symptom(Acetylsalicylic-Acid) = [Nausea, Increase-Sweating,
Elevation-Of-Temp, Insomnia, Headache, Irritability, Excitability,
Muscle-Tremor, Increase-Heart-Rate, Headache, Insomnia,
Irritability, Excitability]
Symptom(Aspirin) = [Nausea]
Symptom(Aspirin-And-Caffeine) = [Muscle-Tremor, Nausea,
Headache, Insomnia, Irritability, Excitability, Muscle-Tremor,
Increase-Heart-Rate]

*Never asked questions:
Age(Victim)

DONE WITH Validator.
```

Figure 2. Validator's output for the Poison Control Expert System.

- *Legal values* are acceptable answers to questions.
- *Utilized values* allow the rule premise in which they appear to be evaluated to true.
- *Concluded values* appear in rule conclusions and are set for an expression when a rule using that expression succeeds.
- *Assigned values* are assigned to expressions with facts or certain commands specific to the production language.

In a simple knowledge base, the sets of legal, utilized, concluded, and assigned values may be the same, but this is not always the case. In Figure 3, the set of legal values for the expression Location is [Around-Mouth, Between-Fingers, ..., Upper-Extremities]. The set of utilized values is [Feet, Posterior-Pharynx]. The set of concluded values for the expression Rash-Identity is [Athletes-Foot, Viral-Stomatitis].

Out-of-range values: utilized versus legal values. Providing legal values guards against typographical errors and helps in abbreviating long answers. If no legal values were provided, the system would take any user's response as an answer, so typographical errors and wrong responses might escape detection. Even if detected, recovery is hard for a user. Effective error-recovery procedures are time-consuming

```
Question(Location) = "Where on the body
is the rash located?".

LegalValues(Location) = [Around-Mouth,
Between-Fingers, Follows-Nerve-Root,
Genitals, Gingiva, Lower-Extremities,
Palms, Pharynx, Scalp, Soles, Site-Of-
Previous-Injury, Trunk, Upper-Extremities].

If Type = Scaly
And Location = Feet
And ..
Then Rash-Identity = Athletes-Foot cf 90.

If Type = Vesicular
And Location = Posterior-Pharynx
Then Rash-Identity = Viral-Stomatitis cf 95.
```

Figure 3. Utilized, concluded and legal values.

```
If My-Hand = Pair-Queens
And Open-Advice = Pass
Then Stay-Advice = Get-Out.

If My-Hand = Three-Of-A-Kind
And ...
Then Open-Advice = Sandbag.

If My-Hand = Straight-Flush
And ....
Then Open-Advice = Open.

If My-Hand = Garbage
And ...
Then Open-Advice = Cannot-Open.
```

Figure 4. Mismatch between utilized and concluded values.

```
Rule-12:  If Has-Sauce = Yes
          And Sauce = Sweet
          Then Best-Sweetness = Sweet Cf 90
          And Best-Sweetness = Medium Cf 40.

Rule-26:  If Best-Sweetness = Dry
          Then Recommended-Sweetness = Dry.

Rule-27:  If Best-Sweetness = Medium
          Then Recommended-Sweetness = Medium.

Rule-28:  If Best-Sweetness = Sweet
          Then Recommended-Sweetness = Sweet.
```

Figure 5. Mismatch between utilized and concluded values.

```
Goal = Accumulate(Sem).

Rule-1: If Semester(Sem) = Yes
        Then Semdel(Sem) = Done.

Rule-2: If Semdel(Sem) = Yes
        Then Accumulate(Sem) = Yes.
```

Figure 6. Mismatch between utilized and concluded values.

and sharply increase the size of the knowledge base. The developer must build procedures that will backtrack to virtually every branch of the knowledge base tree affected by the wrong response, allow the user to recover from the error, and then return to the same place in the knowledge base before the error was discovered.

However, providing legal values will not prevent the knowledge engineer from using an out-of-range value in a rule.[9] Mycin-derived production languages do not check the knowledge base to ensure that only legal values have been used; these languages only check user responses to questions to see if they match legal values specified by the knowledge engineer. Legal values are related to questions, not to rules or facts. Illegal values are common in knowledge bases and usually force a rule using such a value to fail.

In Figure 3, the two rules for concluded values for Rash-Identity will fail because of illegal utilized values. The second premise of the first rule (Location = Feet) and the second premise of the second rule (Location = Posterior-Pharynx) have utilized values that are not in the legal value set. Therefore, these rules can never succeed. When seeking values for Location, the user will be asked the location of the rash. If he or she tries to answer Feet or Posterior-Pharynx, the production language will refuse the answer. Furthermore, all the rules that use the conclusion of the first rule Rash-Identity = Athletes-Foot as a premise will also fail. Such errors will also stop the system from seeking the rest of the premises that come after the false premise of the rule.

Out-of-range values: utilized versus concluded values. The first rule in Figure 4 will fail due to the mismatch between the utilized and concluded values of the expression Open-Advice. The set of concluded values is [Sandbag, Open, and Cannot-Open]; the set of utilized values is Pass. For the second premise of the first rule to evaluate true, and consequently for the rule to succeed, the expression Open-Advice has to be assigned the utilized value Pass, which is not in the set of concluded values for the expression; hence the rule will always fail.

Figure 5 shows another rule that can never succeed, this time from an old, well-polished, professional expert system (the Wine Advisor). From rule 12, the expression Best-Sweetness will be instantiated either to the value Sweet with certainty factor 90 or to the value Medium with certainty factor 40 (sweet and medium are the concluded values for the expression). For rule 26 to succeed, the Best-Sweetness has to be instantiated to the utilized value Dry. But the conclusion of rule 12 is the only place in the knowledge base where Best-Sweetness can get its values, so rule 26 always fails. (Remember, Validator does not detect mistakes, it only points out potential errors. In this case, the knowledge engineers would surely explain this mismatch by saying, "Rule 26 is there for completeness and possible future expansion.")

Figure 6 shows a serious error: Rule 2 always fails because the expression Semdel(Sem) has a mismatch between its utilized value in the second rule (Yes) and its concluded value in the first rule (Done).

Unused legal values. The syntactic error checker searches the premises of rules looking for declared but unused legal values, and flags them as potential errors. It also lists all unasked questions. Unused legal values are common in knowledge bases. They usually suggest errors or remnants of old constructs that were mistakenly put into the knowledge base by the knowledge engineer, or incompletely removed. Deleting such constructs cuts the size of the knowledge base and speeds inference during a consultation.

```
Question(Restart) = [NL, "Which of the following ... see again ?
    1. Print your consultation,
       .
       .
       .
    8. Restart consultation completely,
    9. QUIT this consultation.
**** Choose one or more of the above ...", NL].

LegalValues(Restart) = Integers(0, 9).
```

Figure 7. A typographical error.

In Figure 7, the knowledge engineer used Integers instead of Integer to inform the system about the set of legal values for the question about Restart. This typographical error caused the syntactic error checker to flag every utilized value of the expression Restart and to state that the string Integers (0,9) was an unused legal value for Restart.

When Validator points out such potential errors, it often prompts the knowledge engineer to change the structure. For example, after seeing that the legal values Normal and Subnormal are flagged as unused legal values for the expression Temperature (see Figure 8), the knowledge engineer might ask the same question more directly, providing fewer options for the user to choose from.

```
Original structure
  Question(Temperature) = "What is the temperature of your patient?
  1. Elevated (greater than 100 degrees Fahrenheit)
  2. Normal (between 98.2 and 99.9 degrees Fahrenheit)
  3. Subnormal (less than 98.2 degrees Fahrenheit)".

  LegalValues(Temperature) = [Elevated, Normal, Subnormal].

Suggested structure
  Question(Temperature) = "Is the temperature of your patient more than 100 degrees
     Fahrenheit?".

  LegalValues(Temperature) = [Yes, No].
```

Figure 8. Unused legal values.

The debugger

Debugging is the tedious, difficult, time-consuming, and costly process of finding and correcting errors in the knowledge base. These errors are usually discovered during developmental testing, and finding them depends on the knowledge engineer's intuition, experience, and common sense. However, computer-aided debugging tools are more reliable because they reduce the possibility of human errors. The Validator debugger checks for rules that use variables, negations, and unknowns. It regards the knowledge base as a closed world, that is, it assumes that all the information about the domain is captured in the knowledge base. This means that all the rules and axioms that can possibly hold are either implied or implicitly modeled in the knowledge base. (The use of variables, negations, unknowns, and the closed-world assumption are all aspects of Prolog-type systems.)

Variable-unsafe rules. Variables can be used in both the premises and the conclusions of rules. They act as symbolic place holders. Each construct that uses variables is logically equivalent to the large set of constructs that could be obtained by replacing these variables with the suitable terms. This violates the spirit of the closed-world assumption. A backward-chaining rule that has variables is considered to be variable-safe (closed) if

(1) Its conclusion is homomorphic to a fact or a consequence in the knowledge base.
(2) Variables that appear as attributes of an expression in a conclusion also appear as attributes of an expression in the rule's premise.
(3) Variables that appear as attributes of an expression in a premise also appear as utilized values in previous premises of the same rule or as attributes of an expression in the conclusion.
(4) Variables that appear as concluded values of a rule also appear as utilized values in the rule's premise.

A knowledge base is variable-safe if its set of rules evaluate to true, no matter what values are substituted for the variables. Variable-unsafe rules usually lead to infinite loops that violate the closed-world assumption. In Figure 9, the variable *X* will be instantiated whenever the rule's conclusion falls in the search path of a goal or a subgoal. However, the safety of the rule is violated by the presence of *Y* in the third premise. In Figure 10, the debugger has flagged two variables *P*1 and *P*2 as attributes that will lead to a variable-unsafe rule. (Further investigation of the rule shows that it is not a dead rule. In fact, it is in the search path of more than one goal tree. The second line shows that the knowledge engineer has commented out a premise necessary for the successful instantiation of the variables *P*1 and *P*2, and for the normal progress of a consultation.)

```
If Air-Conditioner = Air-Conditioner-X
And Type(Air-Conditioner-X) = Central
And Not(Maintained(Air-Conditioner-Y))
Then Maintain(Air-Conditioner-X).
```

Figure 9. A variable unsafe rule.

Illegal use of negations and unknowns. Negations can only be used in the premises of rules. A rule that has a negation in its conclusion violates the closed-world assumption, since a false fact is not explicitly declared in the knowledge base, but rather inferred from not being able to conclude the given value for an expression. Figure 11 shows illegal use of negation.

The most common origin of Unknown is a user's response to a question. Unknown can be concluded as a result of the unsuccessful firing of all the rules that concern an expression. However, like negations, Unknown can only be used in the premises of rules. A rule that concludes Unknown

```
If Course_To_Delete = DepNum
/* And Attribute(DepNum) = [T, P1, P2, U, SG, G, Area] */
And Semester(Sem, Course) = [Depnum, U]
And Resetting(Sem, Course) = Done
.
.
.
And PermDelA(DepNum, P1)
And PermDelB(DepNum, P2)
And Display(DepNum, "Was Deleted From Semester Number", Sem)
Then Delete_Course = Deleted.
```

Figure 10. Another variable unsafe rule.

```
If Type(Heater, X) = Gas
Or Type(Heater, X) = Solar
Then Not(Turn-Off(Heater, X)).
```

Figure 11. An illegal usage of negation.

```
If Name(Subdivision) = "Casas Adobes"
And Age(Home) = Newer
Then Value(Home) Is Unknown.
```

Figure 12. An illegal usage of Unknown.

for an expression violates the closed-world assumption because an unknown fact is not explicitly declared in the knowledge base; rather, it is inferred from not being able to conclude a value for an expression. Figure 12 shows an illegal use of Unknown.

VALIDATOR EVOLVED DURING the development of the Salt River Project Residential Expert System[14] and was tested on 70 expert-system knowledge bases, including senior-, graduate-level, and masters projects at the University of Arizona, systems produced as part of research grants, and demonstration systems provided by commercial software houses. Tables 1-5 show the different types of errors detected by Validator. The syntax analyzer errors are under the category "syntactic errors," the syntactic error checker errors are under "referential integrity," and the debugger errors are under "anomalies" and "unused values." (The other errors in these tables were detected by Validator modules discussed elsewhere.[15])

Validator has found many errors, but it can't find them all; for example, it cannot detect circular and recursive rules, compound structures such as frames or object data types and their anomalies, conflicting rules, or redundancies inside rules.

Some of the methods presented here originated in integrity and completeness checking for database systems. The knowledge base dictionary, for example, is rele-

Table 1. Errors detected in masters projects and precommercial systems.

System	Size (Kbytes)	Syntactic errors	Unused rules	Anomalies	Unused questions	Unused values	Multiple questions	Multiple methods	Referential integrity
Advice	82		3		9			35	1
Chromie	100		1						
Cogito	126		5	25			6	7	
Fundeye	60								
Helper	50				1				14
SRPRES	120		2	2					1
Stutter	81		12		5		1		8

Table 2. Errors detected in student-generated systems (Fall 1988).

System	Size (Kbytes)	Syntactic errors	Unused rules	Anomalies	Unused questions	Unused values	Multiple questions	Multiple methods	Referential integrity
Car	6							1	
Citsys	54				9	6	4		
Dell	55		2	3				3	
Diet	15		2				1		1
Drug	41					17			5
Fever	54		1		3				
Haa	27	2	2	1					1
Ky1	9								
Macxprt	10								1
Napsx	28		1				1		2
Oring	37		1						
Pimonc	28	2							
Soundf	32								1
Volcan	18							1	2
Wire	19	1							

Table 3. Errors detected in student-generated systems (Fall 1987).

System	Size (Kbytes)	Syntactic errors	Unused rules	Anomalies	Unused questions	Unused values	Multiple questions	Multiple methods	Referential integrity
Baja	17						2		
Cook	53								8
Hypotens	21								
Job	23		7	6	3	4			9
Labd	6				2				
PDP11	42		4		3	2			3
Pdoc	20								
Patho	19	2							1
Phone	10								
Patutor	40								
Statcon	13		1						
Slick	20	1	2	1	1	2		3	1
Ungrad	28		1					12	3
VW	27	1		1		2			2

Table 4. Errors detected in student-generated systems (Fall 1986).

System	Size (Kbytes)	Syntactic errors	Unused rules	Anomalies	Unused questions	Unused values	Multiple questions	Multiple methods	Referential integrity
AME	30	1	1						
AMI	52	1							
Automech	102					6		1	
Barman	14	1							
ECU	19								
Editor	6								
Expert	54								
Friz	26								
Health1	35	1	1						3
Health2	62				4	10			
Modex	3					5			1
Motor	19								
Plant	46		2			5			5
Poison	59								
Poker	23	4				2			2
Qaci	38		3	1		5			1
Rashdec	23	1			4			1	6
Readines	16								

Table 5. Errors detected in selected student-generated systems (Fall 1985).

System	Size (Kbytes)	Syntactic errors	Unused rules	Anomalies	Unused questions	Unused values	Multiple questions	Multiple methods	Referential integrity
Animal	4								
Autism	15								
Autism2	16	1					1		
ChromieO	13				9			19	7
Labdes	15		1			9		1	3
Solar	7					2			1
StutterO	8	1				2			

vant to the database dictionary, and the mismatch between utilized and concluded values is an extension of the referential integrity problem in relational database systems. A rule in a knowledge-based system can be considered as an extension of a "view" in the relational data model, so checking the safety and completeness of a rule has its origin in checking for view integrity and consistency in database systems.

Validator brings potential errors to the attention of the knowledge engineer as output messages with the subsequent explanation listing the expression, its value, and the reason for suspecting an error. Some of these errors have serious effects on the knowledge base, such as halting a consultation or preventing a subset of the knowledge base from being active. Other potential errors, such as screening the usage of some clauses, have no effect as long as the knowledge engineer knows exactly what is being done.

References

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

2. S.J. Andriole, *Software Validation: Verification, Testing, and Documentation,* Petrocelli Books, Princeton, N.J., 1986.

3. R.A. DeMillo et al., *Software Testing and Evaluation,* Benjamin Cummings, Menlo Park, Calif., 1987.

4. J.H. Gallier, *Logic for Computer Science,* Harper & Row, New York, 1986.

5. H.J. Levesque, "The Logic of Incomplete Knowledge Bases," in *On Conceptual Modeling,* M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, eds., Springer-Verlag, New York, 1984.

6. P.G. Politakis, *Empirical Analysis for Expert Systems,* Pitman Advanced Publishing, Boston, 1985.

7. K. Parsaye, "Acquiring and Verifying Knowledge Automatically," *AI Expert,* Vol. 3, No. 5, May 1988, pp. 48-63.

8. A.D. Preece and R. Singhal, "Verifying and Testing Expert-System Conceptual Models," *Proc. 1992 IEEE Conf. on Systems, Man, and Cybernetics,* IEEE Press, Piscataway, N.J., 1992, pp. 922-927.

9. T.A. Nguyen et al., "Knowledge Base Verification," *AI Magazine,* Vol. 8, No. 2, Summer 1987, pp. 69-75.

10. R.A. Stachowitz, J.B. Combs, and C.L. Chang, "Validation of Knowledge-Based Systems," *Proc. Second AIAA/NASA/USAF Symp. Automation, Robotics, and Advanced Computing for the National Space Program,* Am Inst. of Aeronautics and Astronautics, New York, 1987, pp. 1-9.

11. B.J. Cragun "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, May 1987, pp. 633-648.

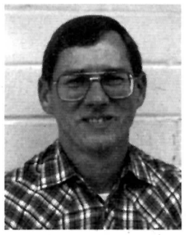
12. M.J. Jafar, *A Tool for Interactive Verification and Validation of Rule-Based Expert Systems,* doctoral dissertation, Univ. of Arizona, Tucson, Ariz., 1989.

13. R.M. O'Keefe, O. Balci, and E.P. Smith, "Validating Expert-System Performance," *IEEE Expert,* Vol. 2, No. 4, Winter 1987, pp. 81-90.

14. M.J. Jafar, A.T. Bahill, and D. Osborn, "A Knowledge-Based System for HVAC," *ASHRAE, J. Am. Soc. Heating, Refrigeration, and Air Conditioning,* Vol. 33, No. 1, Jan. 1991, pp. 20-26.

15. M.J. Jafar and A.T. Bahill, "Verification and Validation with Validator," in *Verifying and Validating Personal-Computer-Based Expert Systems,* A.T. Bahill, ed., Prentice Hall, New York, 1991, pp. 71-83.

Musa J. Jafar is an assistant professor of management information systems in the Department of Management and Systems at Washington State University. He received a BS and MS in mathematics, an MS in systems engineering, and a PhD in systems and industrial engineering from the University of Arizona. He can be reached at the Dept. of Management & Systems, Washington State University, Pullman, WA 99164-4726; Internet, musa@wsuaix.csc.wsu.edu



Terry Bahill is a professor in the Systems and Industrial Engineering Department of the University of Arizona. He is a fellow of the IEEE. Bahill received a BS in electrical engineering from the University of Arizona in 1967, an MS in electrical engineering from San Jose State University in 1970, and a PhD in electrical engineering and computer science from the University of California at Berkeley in 1975. He can be reached at the Systems and Industrial Engineering Department, University of Arizona, Tucson, AZ 85721; Internet, terry@tucson.sie.arizona.edu