From *Operations Research and Artificial Intelligence: The Integration of Problem-Solving Strategies*, (Eds) D.E. Brown and C.C. White, Kluwer Academic Publishers, Boston, pp. 373-385, 1990.

Validator, A Tool for Verifying and Validating Personal Computer Based Expert Systems

Musa Jafar
and
A. Terry Bahill
Systems and Industrial Engineering
University of Arizona
Tucson, AZ 85721
musa@tucson.sie.arizona.edu
terry@tucson.sie.arizona.edu
(602) 621-6561

ABSTRACT

The most difficult tasks in expert system design are verification, validation and testing. Traditional techniques for these tasks require the knowledge engineer to work through the knowledge base and the human expert to run many test cases on the expert system. This consumes a great deal of time and does not guarantee finding all mistakes. On the other hand, brute force enumeration of all inputs is an impossible technique for most systems. Therefore, we have developed a general purpose tool to help verify and validate knowledge bases with little human intervention. Our tool, named *Validator*, has four main components: (1) a Syntactic Error Checker, (2) a Debugger, (3) a Rules and Facts Validation Module, and (4) a Chaining Thread Tracer. It was designed for knowledge bases that use the M.1[1] expert system shell; however, the principles should generalize to any rule-based, backchaining shells, i.e. MYCIN derived shells.

[1] Contrary to popular belief in the AI community, M.1 is still sold, supported and updated by Cimflex Teknowledge Inc.

INTRODUCTION

There are many steps in the process of making an expert system: identifying an appropriate problem domain, learning about the problem domain and the structure of the problem, specifying the input-output performance criteria, selecting a good expert, selecting an expert system shell (or perhaps selecting a language and a quantitative technique for dealing with uncertainty), extracting the knowledge from the expert, encoding this knowledge in the knowledge base, verifying the knowledge base, validating the system, testing the system, updating and maintaining the system, and finally, at the end of its life cycle, retiring and replacing the system (Lehner and Adelman, 1989). This paper discusses verification and validation.

Validation means building the right system: that is writing specifications and checking preformance to make sure that the system does what it is supposed to do. Verification means building the system right: that is ensuring that it correctly implements the specifications. Testing means running test cases on the system to see if it emulates human input-output behavior. In a typical expert system verification is done first, then validation, and finally testing.

Verification is the process of assuring completeness, consistency and correctness of the syntax of a knowledge base. There are many subtasks in verification, they must be performed in the correct order. First we proof read the knowledge base. *Validator* aids the proof reading process by displaying lists of all possible premises, all possible conclusions, all possible facts, all possible legal values for each object, and all possible goal statements. Next we run a spelling checker on it. Then we look for low level syntactic mistakes (most shells will do this function). Then we look for more subtle syntactic mistakes and finally we run our Debugger.

After verification is complete we can do validation. Validation is the process of assuring the compliance of system performance with the specified system requirements and needs. In other words, validation checks the semantics of the system. However, verification and validation of a system does not mean that the system is adequate. Verified and validated systems can still exhibit unsatisfactory functional performance due to poor hardware/software design, poor system-system and human-system interfaces, poor explanation facilities or incomplete and unclear requirements specifications.

After a system is verified and validated, the human expert should run a few dozen test cases through the expert system to test it. It is important to do verification and validation before the testing, to minimize squandering the expert's time finding simple mistakes.

It seems that more mistakes would be detected if many experts tested the system. It is often possible to get an expert to devote a substantial amount of time to a project; being interviewed, verifying and validating the knowledge base, and running test cases. However, we have found that it is difficult to get other experts to devote time to testing the final product for the following reasons: their time is expensive, there are few of them in any geographical area, they might disagree on the criteria used to draw conclusions or even the conclusions themselves, finally, they do not have the personal commitment to the project to compel them to donate copious amounts of time. Therefore, it is difficult to get multiple experts to exhaustively test an expert system. So, it is important to maximize the efficiency of utilizing the domain experts.

Therefore, we have built a general purpose tool to help verify and validate knowledge bases without extensive intervention by human experts during the development stages of an expert system. We tried to make this tool generic so that it can work on any rule-based knowledge base no matter which expert system shell is used. The first two components of the system, the Syntactic Error Checker and the Debugger are a part of verification. While the last two components, The Rules and Facts Validation Module and The Chaining Thread Tracer are a part of validation.

There are other programs for verification and validation of expert system knowledge bases: *Teiresias* for the MYCIN system (Davis, 1976), a program for ONCOCIN (Suwa, Scott and Shortliffe, 1982), *Check* for programs written with Lockheed's LES shell (Nguyen, Perkins, Laffey and Pecora, 1987), ARC for production systems written with ART (Nguyen, 1987), and EVA for knowledge bases using ART and LISP (Stachowitz, Combs and Chang, 1987; Stachowitz, Chang, Stock and Combs, 1987). A sixth similar tool is ESC, a decision-table-based processor for checking completeness and consistency in rule-based expert systems (Cragun, 1987). In contrast to these programs, our system, *Validator*, is specifically designed to run on personal computers.

VALIDATOR

The Syntactic Error Checker

The first component of *Validator* is the Syntactic Error Checker. Syntactic errors are common in expert systems knowledge bases; many of these are misspellings or typographical errors. The expert systems submitted by graduate and undergraduate students as class projects for our Expert Systems course in the Fall of 1987 had an average of 12 spelling errors per knowledge base. The Syntactic Error Checker also checks for syntax that, although legal, produces unspecified behavior of the system such as (1) the use of *is known* or a negation in the conclusion of a rule, e.g.

then type(air-conditioner) is known,
then not(type(air-conditioner) = central);

(2) negations in the right hand sides of premises, e.g. *type(air-conditioner) = not(central)*, where the knowledge engineer probably wanted *not(type(air-conditioner) = central)*, and (3) instantiating objects to *known, unknown, found* or *sought* such as *type(air-conditioner) = known*, where the knowledge engineer probably wanted to use the metafact *is*, e.g. *type(air-conditioner) is known*. This component also checks each user defined object, attribute and value to make sure that it is not a reserved word such as *or, and, mod, add, is,* or *off*.

Providing legal values for an object ameliorates typing errors in response to a question. If no legal values were provided, the system would take any user's response as an answer. So typographical errors might escape detection, even if they were detected, it is hard for a user to recover without restarting the whole system. Providing legal values also allows the user to abbreviate his response, for example, he can type *y* instead of *yes* as an answer. Another type of syntactic error occurs when an illegal value is written into a knowledge base. Most shells check user responses to questions to see if they match legal values specified by the knowledge engineer. However, they do not check the rules to ensure that only legal values have been used. Legal values are associated with questions, not with rules or terms. In figure 1, rule 4 shows an example of a rule using an illegal value. Premise 2 of rules 3 and 5 will similarly be flagged, because no legal values were provided for *marking of animal*. Figure 1 also shows an unused legal value, the value *claws* for the object *extremities*

of animal was never used in the rule base. In this section, we have shown examples of 7 of the 10 syntactic errors that *Validator* can detect. More examples are given in Jafar (1989), and Bahill (1990).

The Debugger

It has been estimated that testing and debugging comprised 80 percent of the cost of the NASA Apollo project (Yourdon, 1975), 44 percent of the cost of the Saturn 1 project, and 50 percent of the cost of the Naval Tactical Data System (Boehm, 1970). Manual debugging of a knowledge base is expensive and time consuming, it is also difficult, error prone and does not guarantee the finding of all bugs. Debugging means removing compiler specific mistakes that cause the system to not compile or to fail at run time. In the following rule, the variable X will be instantiated whenever the conclusion of the rule falls in the search path of a goal. However, the knowledge engineer mistakenly typed a Y instead of an X in the third premise. If this rule is encountered during a consultation the computer will halt, because the variable Y can not be instantiated.

```
if air-conditioner = air-conditioner-X
and type(air-conditioner-X) = central
and not(maintained(air-conditioner-Y))
then maintain(air-conditioner-X).
```

Validator also checks for unclosed comments. Shells and compilers can detect an unclosed comment if it is the last comment in the knowledge base, but if the unclosed comment is followed by another comment, the unclosed comment will probably escape detection. An unclosed comment warning is issued whenever the Debugger encounters a second beginning of a comment string /* without closing the first one. This type of error is typographical. It usually forces the inference engine to ignore the part of the knowledge base that lies between the two comments.

Debugging also includes the actions of a knowledge engineer running the system and watching for aberrant behavior, such as unexpected questions being asked. Interactive debugging removes many inconsistencies that are virtually impossible to detect manually by a knowledge engineer.

The Rules and Facts Validation Module

When verification is complete and all the syntactic errors are removed from the knowledge base, then validation can begin. Validation means ensuring that the system does what it was supposed to do. Rules that can never fire are typical of mistakes that can be detected by our Rules and Facts Validation Module. For example Rule 6 of figure 1 shows such a rule that can never fire. This module also checks rules and facts for validity. For example the next rule will always fail because of the declared fact.

fact: prob-cards = no.

rule: if found-tag = yes
and prob-cards = yes
then lost-tag = yes.

The object prob-cards will never be instantiated to the value *yes*, because it was set to the value *no* by a fact. The above example is from the Carpet Advisor, one of the student generated systems.

The Chaining Thread Tracer

The fourth and last component of *Validator* is the Chaining Thread Tracer. It was designed for backchaining shells, but was later enlarged to handle forward chaining constructs (Jafar, 1989). It checks the validity of each rule by tracing it's premises and conclusions to see if they are properly interconnected. We call this component a Chaining Thread Tracer, because it traces connectivity of the terms through the backchaining system.

Backward chaining systems start their search with a goal as the root of an inverted tree and rules as branches. For example, the goal *identity of animal* of figure 1 brings us to the conclusion of rule 3. From here the term *subtype of animal* links the first premise of rule 3 to the conclusion of rule 2. Next the term *type of animal* links the premise of rule 2 to the conclusion of rule 1. The conclusion of rule 1 is then linked to the premise of rule 1, *coat of animal = hair*. There is a question that can provide a value for this term, coat of animal, so this chain of linking is good.

To detect logical errors in backchaining systems, *Validator* checked every premise of every rule to make sure that it led to a valid end. A premise has a valid end if it appears in the conclusion of another rule or a question is provided for it by the knowledge engineer. For example, in figure 1 the first premise of rule 2 is a valid end, since it appears as a conclusion in rule 1, the second premise of rule 2 is a valid end also, since a question was provided for it. We also checked the conclusion part of every rule for valid ends. A rule's conclusion is considered to be a valid end if it is either a goal statement or it appears as a premise in another rule. For example, in figure 1 the conclusion of rule 1 appears as a premise in rule 2, in turn, the conclusion of rule 2 appears as a premise of rule 3, the conclusion of rule 3 is a goal. Hence rules 1, 2, and 3 all lead to valid ends. An added complexity in the conclusion validation checking that we have to deal with is caused by the fact that, conclusions of rules can be linked with two types of premises. A rule with a conclusion of the form *then animal = mammal*, has to be linked with the premise *if animal = mammal* and also with *if not(animal = ANY)*, where *ANY* can take any value other than mammal, since both premises are going to be in the search tree of a goal statement.

Figure 1 shows a typical knowledge base. The first three rules are correct. If you have a zebra in mind, it will correctly identify this animal. The rest of the rules illustrate various mistakes that might result when a knowledge base is expanded. Rule 6 is extraneous, it will never be reached during inferencing. Rule 5 shows a mistake that the Chaining Thread Checker will detect. There is no way to get a value for the object *feed of animal*. The object has no question and it does not appear in the conclusion of any other rule. Therefore, it will be flagged as a potential error.

```
infix of.
goal = identity of animal.
rule1:   if coat of animal = hair
         then type of animal = mammal.
rule2:   if type of animal = mammal  and
         extremities of animal = hooves
         then subtype of animal = ungulate.
rule3:   if subtype of animal = ungulate and
         marking of animal = black-stripes
         then identity of animal = zebra.
rule4:   if coat of animal = scales
         then identity of animal = fish.
rule5:   if feed of animal = meat and
         marking of animal = black-stripes
         then identity of animal = tiger.
rule6:   if coat of animal = feathers and
         animal-swims
         then habitat of animal = antarctic.
question(coat of animal)=('What is the coat of animal?').
legalvals(coat of animal)=[hair, feathers].
question(extremities of animal) =
('What is type of the extremities of the animal?').
legalvals(extremities of animal) = [hooves, claws].
question(marking of animal) =
('What is type of the marking of the animal?').
```

Figure 1. A small knowledge base with typical errors. The infix operator is just a convenient way of saving space and improving readability. It allows us to say "coat of animal" instead of "coat-of-animal".

Results of Testing Validator

Over a period of three years, 50 student generated expert systems were used in the development of *Validator*. Another 14 new expert systems, which were created as final projects by students in our Fall 1988 Senior/Graduate student Expert Systems course, were then used to test it. Each of these programs took an average of 100 hours to create and an average of 50 Kbytes of disk space. Table 1 summarizes some of the potential errors detected by *Validator* in these systems. Rule 4 of figure 1 is an example of a rule using an illegal value. Figure 1 also shows an unused legal value *claws* for the object *extremities of animal*. Rule 6 in figure 1 is an example of a rule that can never fire. On the next page we show an example of a rule (rule-26) that can never succeed.

Table 1. The number of certain types of potential errors detected by *Validator* in 14 student generated expert system knowledge bases.

System Name	Rules using illegal values	Unused legal values	Use of reserved words	Unused rules
Veterinarian	0	9	0	0
Automech	0	0	0	0
Carpet	2	29	0	0
CompSel	1	10	0	1
Diet	1	4	0	2
Legal Drug	5	19	0	0
HAA	0	7	1	2
MacExpert	0	2	3	0
NAPSX	2	36	0	1
O-ring	0	60	0	0
Software	0	3	0	1
SoundFilm	0	72	0	0
Volcanic	2	10	0	0
Wire	0	3	3	0

Validator sometimes flagged items that were not mistakes, but rather were handled by the knowledge engineer outside the expert system. For example O-ring and SoundFilm had external data collection programs that

caused *Validator* to think that there were errors where such errors probably did not exist. After testing *Validator* with these 14 class projects, we tested it with five more advanced expert systems. The results of these tests are shown in Table 2.

Table 2. Number of potential mistakes detected by *Validator* in five more advanced expert systems.

System Name	Rules using illegal values	Unused legal values	Use of reserved words	Unused rules
Advice	1	1	0	3
Helper	2	1	0	0
Stutter	7	5	0	6
Fund-Eye	0	2	1	0
Wine	0	2	0	1

The first three systems, Advice, Stutter and Helper, were master's theses projects in the Department of Systems and Industrial Engineering at the University of Arizona. Advice was designed to recommend a study plan for new graduate students. Helper was designed to aid students using the computer laboratory. Stutter was designed to help with the diagnoses and prognosis of children who may have begun to stutter. The fourth system, Fund-Eye, is a retinal disease diagnostic system. The fifth system, Wine, is a demonstration program provided by Teknowledge (the developer of M.1). The following set of rules, taken from the Wine Advisor, shows a rule that can never succeed.

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.

From rule-12, the object *best-sweetness* will be instantiated either to the value *sweet* with certainty factor 90 or to the value *medium* with certainty factor 40. For rule-26 to succeed, the object *best-sweetness* would have to be instantiated to the value *dry*. But this can not happen, because the conclusion of rule-12 is the only place in the knowledge base where *best-sweetness* gets a value. Therefore rule-26 can never succeed.

Once again we note that *Validator* flags potential mistakes. It is up to the knowledge engineer to decide if it is an actual mistake or not. For example in the above example from the Wine Advisor it is obvious that the knowledge engineer wanted to include rule-26 for completeness, although it would never fire. These tables show data for only 5 out of the 17 types of potential errors that *Validator* currently detects. In previous sections we showed examples of another 8 types of potential errors that it detects.

Undetected Errors

We could continue enlarging *Validator*, but it would never be able to detect all possible errors. For example, the following constructs were meant to allow the user to change his mind about his last answer, and backup to correct it:

whencached(X=Y) = [(not(Y=oops)) and temp=X].
whencached(X=Y) = [(Y=oops) and (do(reset X), do(reset temp), temp)].

However, what resulted was circular reasoning; it put the system into an infinite loop. We did not enlarge *Validator* to detect this particular error.

SUMMARY

Ideally we want to build expert systems that pass exhaustive tests and are accepted by all experts. This goal is not reachable. We will never be able to guarantee that a system works exactly as intended or that it fully meets its requirements specifications. It is almost impossible to generate test cases that exhaust all rules. So some rules are going to escape the verification, validation and testing. Tables 1 and 2 show that *Validator* guarantees that each rule is checked for validity and consistency. It also guarantees that the rule will fire whenever enough evidence is gathered to satisfy its premises. We developed verification and validation tools to find out why a system is not working right and to increase the confidence in the level of performance of a system.

ACKNOWLEDGEMENT

Validator can be purchased from the authors. Research of this paper was partially supported by Grant Nr. AFOSR-88-0076 from the Air Force Office of Scientific Research.

REFERENCES

Bahill A.T. (1990) *Verification and Validation of Personal Computer Based Expert Systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Boehm B.W. (1970) *Some information processing implications of Air Force space missions:* 1970-1980. Memorandum RM-6213-PR, RAND Corp, Santa Monica.

Cragun B.J. (1987) A decision-table-based processor for checking completeness and consistency in rule-based expert systems. *Int. J. Man-Machine Studies* **26**, 633-648.

Davis R. (1976) *Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*, Ph.D. dissertation, Department of Computer Science, Stanford University.

Jafar M.J. (1989) *A Tool for Interactive Verification and Validation of Rule Based Expert Systems*, Ph.D. dissertation, Department of Systems and Industrial Engineering, University of Arizona.

Lehner P.E. and Adelman L. (1989) (Eds.) Special issue on perspectives in knowledge engineering. *IEEE Trans. Syst. Man Cybern.* **SMC-19,** 433-662.

Nguyen T.A. (1987) Verifying consistency of production systems. In *Proc. of IEEE Third Conf. on AI Applications* pp. 4-7. IEEE, New York.

Nguyen T.A., Perkins W.A., Laffey T.J. and Pecora D. (1987) Knowledge base verification. *AI Magazine* 8(2), 69-75.

Stachowitz R.A., Chang C.L., Stock T.S. and Combs J.B. (1987) Building validation tools for knowledge-based systems. In *Proc. of First Annual Workshop on Space Operations Automation and Robotics* pp. 209-216. Houston, Texas.

Stachowitz R.A., Combs J.B. and Chang C.L. (1987) Validation of knowledge-based systems. In *Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, pp. 1-9. American Institute of Aeronautics and Astronautics, New York.

Suwa M., Scott A.C. and Shortliffe E.H. (1982) An approach to verifying completeness and consistency in a rule base. *AI Magazine* 3(4), 16-21.

Yourdon E. (1975) *Techniques of Program Structure and Design.* Prentice-Hall Inc., Englewood Cliffs, New Jersey.