N89-19834

Testing Expert Systems

C. L. Chang and R.A. Stachowitz
Lockheed Missiles & Space Company, Inc.
Lockheed Artificial Intelligence Center, O/90-06, B/30E
2100 East St. Elmo Road
Austin, Texas 78744

Abstract

Software quality is of primary concern in all large-scale expert system development efforts. Building appropriate validation and test tools for ensuring software reliability of expert systems is therefore required.

The Expert Systems Validation Associate (EVA) is a validation system under development at the Lockheed Artificial Intelligence Center. EVA provides a wide range of validation and test tools to check the correctness, consistency, and completeness of an expert system.

Testing is a major function of EVA. It means executing an expert system with test cases with the intent of finding errors. In this paper, we describe many different types of testing such as function-based testing, structure-based testing, and data-based testing. We describe how appropriate test cases may be selected in order to perform good and thorough testing of an expert system.

INTRODUCTION

It has been repeatedly shown that the expert system technology in artificial intelligence can be used to implement many different applications such as diagnostic systems, battle management systems, machine and robot control systems, monitoring systems, design systems, manufacturing systems, etc. Regardless of whether the expert systems are stand-alone or real-time embedded systems, we need to be ensured that they are reliable, correct, consistent and complete. For this purpose, the Lockheed Artificial Intelligence Center started in 1986 the Expert Systems Validation Associate (EVA) project [Stachowitz et al. 1987a, 1987b, 1987c]. EVA provides a wide range of validation and test tools to check the correctness, consistency and completeness of an expert system.

Testing is a major function of EVA. It means executing an expert system with test cases with the intent of finding errors. A very good example is the Target Generation Facility (TGF) which provides simulated, real-time controllable aircraft targets to Air Traffic Control Systems under test at the FAA Technical Center.

In this paper, we consider many different types of testing such as function-based testing, structure-based testing and data-based testing. We describe how appropriate test cases may be selected in order to perform good and thorough testing of an expert system.

BACKGROUND

Expert systems are usually developed incrementally. The initial requirements for an expert system may be clearly stated. However, as the expert system evolves and is evaluated, the requirements may be changed or new requirements may be added. In many cases, even if the requirements are not changed, there are no known algorithms for solving the problem. For example, there are no algorithms for performing parallel parking even though the initial and final positions of a car can be specified precisely. Therefore, an expert system may have to be developed in repeated cycles of implementation, evaluation and modification steps. In the parallel parking example, a fuzzy (approximate) algorithm, represented by rules, may be tried first. The algorithm continues to be modified until a satisfactory performance is achieved. The goal of the test case generator is to generate "appropriate" test cases from the requirements specifications or the expert system itself for users, expert collaborators, or system builders to perform the thorough evaluations during the development or acceptance phase of the system production cycle.

Testing of conventional software [DeMillo et al. 1981, 1987, Hetzel 1984, Miller and Howden 1984, Zeil and White 1980] has been known for a long time. As stated in [Hetzel 1984], software testing is a creative and difficult task. It requires very good knowledge about the system being tested. Typically, the requirements cannot be processed automatically, or knowledge is buried inside the codes of the system. Therefore, test cases are conventionally generated manually. This is certainly tedious and error-prone.

On the other hand, an expert system is usually implemented in a high-level language that supports high-level concepts such as objects, relations, categories, functional mappings, data types and data constraints. This knowledge can be used to generate test cases automatically.

TYPES OF TEST CASES

In this paper, we consider three types of test cases, namely, *function-based test cases, structure-based test cases, and data-based test cases.*

To generate function-based test cases for an expert system, one requires knowledge about the system's functions. Function-based testing is usually regarded as *black box testing* because it tests the external input-output behavior (specifications) of the system. In order to generate function-based test cases *automatically*, the generator must be provided with knowledge about the input-output specifications.

Structure-based test cases explore the relations between rules. An expert system can be represented by a knowledge base consisting of facts and rules, which can be connected to make a connection graph. An arc in the connection graph denotes a match between a literal in the left hand side (LHS) of a rule and a literal in the right hand side (RHS) of a rule. Note that a fact can be considered as a rule without a RHS. Structure-based test cases are based upon the structure of the connection graph. The idea is to generate a set of test cases to exercise every rule in the connection graph at least once.

The difference between function-based and structure-based testing can be illustrated by using an electrical circuit: Function-based testing means checking whether the light goes on when we throw the switch, while structure-based testing means inspecting whether all parts are connected properly into the circuit. To perform function-based testing, we do not need to look inside the circuit box. Therefore, it is called *black box* testing. On the other hand, since we need to look inside the circuit box to see how parts are connected, structure-based testing is called *white box* testing.

Data-based test cases are based upon data definitions for the expert system. The data definitions consist of data declarations and data constraints. The data declarations are schema statements for data domains, relations and objects. A data constraint is specified by a logic formula using object-level and/or meta-level predicates.

FUNCTION-BASED TESTING

Input data to an expert system are usually represented by facts that are instances of schemas. Let us call these schemas *input schemas*. Each test case contains a set of facts of the input schemas. For each set of input facts, the expert system will produce a set of output facts (data), which are instance of *output schemas*.

The input and output schemas may not be declared *explicitly*. They may be *implicitly* contained in the connection graph of the expert system. In this case, we consider only the rule part of the connection graph. In a connection graph, there are two kinds of leaf nodes, namely, *input* nodes and *output* nodes. An input node is a LHS-literal of a rule that is not connected to other RHS-literals. An output node is a node representing a RHS-literal that is not connected to any LHS-literals. The schemas of input and output nodes will be considered as the input and output schemas, respectively.

In order to thoroughly cover all different types of input test cases, we must systematically categorize input and output data by explicit declarations. For each set of input facts in certain categories, we specify the expected output facts, or the expected categories of the output facts, or the data constraints that the expected output facts have to satisfy.

Consider the airline inquiry system in [Hetzel 1984]. The specifications of the system are given as follows: The inputs are 1) a transaction identifying departure and destination cities and travel date, and 2) tables of flight information showing flights available and seats remaining. The system checks the flight tables for the desired city. If there is no flight to that city, it prints message 1 "No flight". If there is a flight, but seats are not available, it prints message 2 "Sold out". If there is a flight and seats are available, it displays that fact. Therefore, the expected ouput is either a flight display, or message 1, or message 2.

For this example, the relational schema is:

```
flight(flight#, from_city, to_city, date, seats_reserved, capacity)
```

where flight# is a key. The data base contains a collection of ground instances (facts) of the flight relation. To generate test cases, we specify the following categories of flights:

```
category(flight,no_flight(X,Y,D)):-   /* no flight from X to Y */
        A={F# | flight(F#,X,Y,D,_,_)}, count(A)=0.
category(flight,single(X,Y,D)):-   /* single flight from X to Y */
        A={F# | flight(F#,X,Y,D,_,_)}, count(A)=1.
category(flight,multiple(X,Y,D)):-   /* multiple flight */
        A={F# | flight(F#,X,Y,D,_,_)}, count(A) > 1.
category(flight,full(F#,X,Y,D)):-   /* flight F# from X to Y is full */
        flight(F#,X,Y,D,S,C), count(S)=C.
category(flight,available(F#,X,Y,D)):-   /*flight is available */
        flight(F#,X,Y,D,S,C), count(S) < C.
```

Similarly, the categories of the output on a computer display are:

```
category(output,one_line).
category(output,multiple_lines).
category(output,message_1).
category(output,message_2).
```

(Note that we use the Prolog syntax for representing facts and rules, where a variable is written as a string beginning either with a capital letter or "_".)

For each set of input facts (data) belonging to a certain combination of categories, we specify the expected output. For this example, the input-output relationships specified in terms of categories are given as follows:

(1) single & available --> one_line.

(2) multiple & available --> multiple_lines.

(3) no_flight --> message_1.

(4) single & full --> message_2.

(5) multiple & full --> message_2.

Based upon these functional specifications, the test case generator can generate the following test cases to cover different input scenarios:

CASE 1A: Flight available (only flight to the city).
EXPECTED RESULT: Display one line.

CASE 1B: Flight avilable (multiple flights to the city).
EXPECTED RESULT: Display multiple lines.

CASE 2: No flight.
EXPECTED RESULT: Message 1.

CASE 3A: No seats (only flight to the city).
EXPECTED RESULT: Message 2.

CASE 3B: No seats (multiple flights, all full).
EXPECTED RESULT: Message 2.

CASE 4: Flight available (one flight full, but another open).
EXPECTED RESULT: Display lines and Message 2.

Note that each of CASE 1A through 3B corresponds to one of the input-output relationships specified above. However, CASE 4 is generated by using the input-output relationships (2) and (4). This is possible because the conditions in the input-output relationships (2) and (4) are not mutually exclusive.

STRUCTURE-BASED TESTING

Another important source of test cases derives from the structure of a knowledge base, namely, the connection graph. The advantage of structure-based testing is that the generation of test cases depends upon only the connection graph. It does not have to rely upon other information such as input-output specification of the system represented by the knowledge base.

The basic concept in structure-based testing is one of *complete coverage*. The assumption is that every rule in the connection graph in some way serves some purpose for handling certain situations. Therefore, all the rules must be useful, i.e., used some time, and the goal of structure-based testing is to generate a set of test cases to exercise every rule at least once. An algorithm for generating such test cases follows:

(1) Generate the connection graph of the knowledge base.

(2) Generate the rule flow diagram from the connection graph. Note that a *rule flow diagram* is a directed graph where nodes denote rules, and arcs denote rule execution sequences.

(3) Create a set of paths in the flow diagram such that each node (rule) is covered by at least one path in the set.

(4) Generate test cases to traverse these paths.

Consider a rule-based system that computes the grade of a student from his answers to a quiz. The system compares his answers with the expected answers, counts the number of right answers, computes a numerical score, and then records the grade. His answers are represented by *student(Name,Answers)*, and the expected answers are represented by *expect(N_questions, Correct_answers)*. An instance of *student* and an instance of *expect* constitutes an input to the system. The rules for this system are given as follows:

```
(1) grade(Name,Grade):- student(Name,Answers),
        expect(N_questions, Correct_answers),
        right_answers(Answers, Correct_answers, N_rights),
        Ratio is N_rights/N_questions,
        Score is Ratio*100,
        compute_grade(Score,Grade).

(2) right_answers([],[],0).
(3) right_answers([X|Y], [X|Z], R1):-
        right_answers(Y,Z,R),
        R1 is R+1.
(4) right_answers([_|Y], [_|Z], R):- right_answers(Y,Z,R).

(5) compute_grade(Score,a):- Score>=90.
(6) compute_grade(Score,b):- Score<90, Score>=80.
(7) compute_grade(Score,c):- Score<80, Score>=70.
(8) compute_grade(Score,f):- Score<70.
```

The rule flow diagram for these rules is shown in Figure 1. From the rule flow diagram, we can construct, for example, a set of paths, [1,3,4,2,8], [1,3,2,5], [1,4,3,3,3,2,7], and [1,3,4,3,3,3,2,6]. This set has a complete coverage of the rules, because every rule appears in the set at least once. For each path in the set, we collect all the conditions of the rules in the path, and find values that satisfy the conditions. If such values exist, then the path can be traversed, and the values can be used as a test case. The test cases for the paths are shown in Table 1.

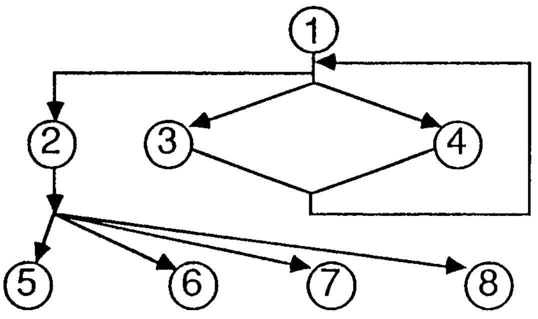

Figure 1. Rule Flow Diagram

TABLE 1. TEST CASES FOR COMPLETE COVERAGE

TEST CASES	*PATHS TRAVERSED*
student(john, [yes,yes]). expect(2, [yes,no]).	1,3,4,2,8
student(smith, [yes]). expect(1, [yes]).	1,3,2,5
student(peter, [yes,yes,yes,no]). expect(4, [no,yes,yes,no]).	1,4,3,3,3,2,7
student(mary, [yes,no,yes,no,yes]). expect(5, [yes,yes,yes,no,yes]).	1,3,4,3,3,3,2,6

DATA-BASED TESTING

We now consider test cases that are derived from data definitions. Such test cases are called *data-based test cases*. Data definitions include data declarations and data constraints. In an expert system shell, data declarations are specified by data schema statements. Since maintaining the integrity of facts and rules in a knowledge base is important, we need also to specify the data constraints that the facts and rules must satisfy. Any fact or rule that violates the data constraint will not be inserted into the knowledge base. We can use logical formulas to represent the data constraints.

By means of the data declarations and data constraints in the expert system, we can generate good and bad test cases. A good test case satisfies the data declarations and data constraints and should be accepted by the expert system, while a bad test case violates them and should be rejected by the expert system. Because the goal is to test the expert system with difficult examples, we should generate some extreme cases that barely satisfy or violate the data constraints, or contain large or small values.

Consider input data on triangles specified by

RELATIONAL SCHEMA:

triangle(side1:number, side2:number, side3:number).

DATA CONSTRAINT:

$$\begin{aligned} &\text{triangle}(X,Y,Z) \wedge \\ &X + Y > Z \wedge \\ &X + Z > Y \wedge \\ &Y + Z > X. \end{aligned}$$

The data constraint says that the sum of any two sides of a triangle is greater than the remaining side. From the above data declaration and data constraint, we can generate the following extreme test cases. (Note that the first five test cases are bad, while the last two are good.)

EXTREME TEST CASES	*COMMENTS*
triangle(1, 1, 2)	A straight line
triangle(0, 0, 0)	A point
triangle(4, 0, 3)	A zero side
triangle(1, 2, 3.00001)	Close to a triangle
triangle(9170, 8942, 1)	Very small angle
triangle(.0001, .0001, .0001)	Very small triangle
triangle(83127, 74326, 96652)	Very large triangle

For an *applicative* system which takes an input and produces an output, a test case means a simulated instance of input and its expected output. However, for an *imperative* system that may alter data structures or produce side effects, just generating test cases of input is not enough. An imperative system can be represented by a state machine. There are a number of states. For each state, there are a certain number of actions that take the state into other states. For the state machine, a test case will be actually a test scenario that consists of an initial state, and a sequence of specific actions. The goal is to check if bad states will be encountered when we run the state machine with the test scenario. We note that a bad state means that the state violates integrity constraints or a situation where no actions are available.

Consider the following example: Container A can hold 5 gallons of water and container B 2 gallons of water. Initially, A is full and B is empty. Assume that water can be poured from A to B, and B to the drain. We would like to get to a final state where A is empty and B is half-full. The initial and final states are shown in Figure 2.

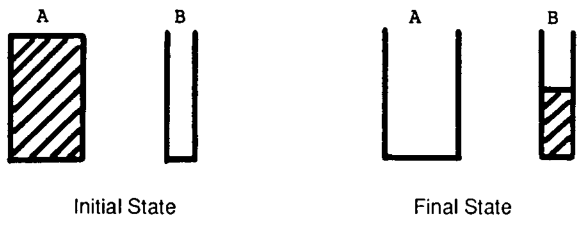

Figure 2. Initial and Final States

We use *state(X,Y)* to denote a state where X and Y are the amounts of water in containers A and B, respectively, and use *pour(X,Y,Q)* to denote an operation to pour Q gallons of water from X into Y.

Let *transition(Op,X,Y)* denote that the operation Op changes state X to state Y, and let *reach(Seq,X,Y)* denote that the sequence of operations, Seq, changes state X to state Y.

The constraints on states and operations are specified as follows:

pour(a,b,X) $\wedge$ $0 < X \leq 2$.

pour(b,drain,X) $\wedge$ $0 < X \leq 2$.

state(X,Y) $\wedge$ $0 \leq X \leq 5 \wedge 0 \leq Y \leq 2$.

From the constraints, we can generate the following test scenario:

state(5,0).	*initial state*
pour(a,b,2), pour(a,b,2).	*input sequence*

This is a bad test scenario because the second operation in the input sequence will cause container B to overflow. If we had the following knowledge base,

```
(1) transition( pour(a,b,Z), state(X,Y), state(U,V) ) :-
        Z>0 ∧
        U=X-Z ∧
        V=Y+Z.

(2) transition( pour(b,drain,Y), state(X,Y), state(X,0) ) :- Y>0.

(3) reach([Op],S1,S2) :- transition(Op,S1,S2).
(4) reach([Op|Seq], S1, S3) :-
        transition(Op,S1,S2),
        reach(Seq,S2,S3).
```

the bad scenario would be "successfully" processed, because Rule (1) is wrong. The correct version of the rule should be

```
(1') transition( pour(a,b,Z), state(X,Y), state(U,V) ) :-
        Z>0 ∧
        U=X-Z ∧
        V=Y+Z ∧
        X≥Z ∧
        V≤2.
```

The correct rule does make sure that container B will not overflow.

CONCLUSION

Test cases of input values to an expert system can be generated automatically. However, the expected output and performance for each test case may not be known, or not clearly defined, or stated in qualitative or narrative statements. In this case, the system's output and performance for the generated (simulated) test cases may have to be evaluated by independent human experts. The experts' evaluation results can be stored and used with the test cases again when the expert system is modified.

We have described systematic ways for automatic test case generation. For large expert systems, this is essential because manual approaches are tedious and possibly biased.

We have started work on implementing components of the test case generator. First, we will generate structure-based test cases because they do not depend upon specifications and metaknowledge. Then, we will consider data-based and finally function-based test cases.

REFERENCES

Bellman, K.L., and Walter, D.O. [1987] "Testing rule-based expert systems", Technical Report, Knowledge-Based Systems Section, Computer Science Laboratory, The Aerospace Corporation, Los Angeles, Ca 90009-2957, November 1, 1987.

Chang, C.L. [1976] "DEDUCE --- A deductive query language for relational data bases", in *Pattern Recognition and Artificial Intelligence* (C.H. Chen, Ed.), Academic Press, Inc., New York, 1976, pp.108-134.

Chang, C.L. [1978] "DEDUCE 2: Further investigations of deduction in relational data bases", in *Logic and Data Bases*(H. Gallaire and J. Minker, Eds.), Plenum Publishing Corp., New York, 1978, pp.201-236.

Chang, C.L. [1981] "On evaluation of queries containing derived relations in a relational data base", in *Advances in Data Base Theory --- Volume 1* (H. Gallaire, J. Minker and J.M. Nicolas, Eds.) Plenum Publishing Corp., 1981, pp.235-260.

Culbert, C., Riley, G., and Savely, R.T. [1987] "Approaches to the verification of rule-based expert systems", *Proc. of First Annual Workshop on Space Operations Automation and Robotics,* Houston, Texas, August 1987.

DeMillo, R.A., Hocking, D.E., and Merritt, M.J. [1981] "A comparison of some reliable test data generation procedures", GIT-ICS-81/08, School of Information and Computer Science, Georgia Institute of Technology, April 1981.

DeMillo, R.A., McCracken, W.M., Martin, R.J., and Passafiume, J.F. [1987] *Software Testing and Evaluation,* The Benjamin/Cummings Publishing Company, Inc., 2727 Sand Hill Road, Menlo Park, California 98025.

Geissman, J.R., and Schultz, R.D. [1988] "Verification and validation of expert systems", *AI Expert*, February 1988, pp.26-33.

Green, C.J.R., and Keyes, M.M. [1987] "Verification and validation of expert systems", *IEEE Knowledge-Based Engineering & Expert System* (WESTEX-87), IEEE 87CH2463-8, 1987, pp.38-43.

Hetzel, W. [1984] *The Complete Guide to Software Testing*, QED Information Sciences, Inc., Wellesley, Massachusetts, 1984.

Miller, E.F. [1987] "Expert system validation: Issues and approaches", in *Expert Systems and Their Applications*, May 1987, Avignon, France.

Miller, E.F., and Howden, W. [1984] *Software Testing and Validation Techniques*, 2nd Edition, IEEE Computer Society Press, 1984.

Jacob, R.J.K., and Froscher, J.N. [1986] "Developing a software engineering methodology for knowledge-based systems", NRL Report 9019, Computer Science and Systems Branch, Information Technology Division, Naval Research Laboratory, Washington, D.C. 20375-5000.

Nguyen, T.A. [1987] "Verifying consistency of production systems", *Proc. of the 3rd IEEE Conference on AI Applications*, February 1987, pp.4-8.

Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D. [1985] "Checking an expert system's knowledge base for consistency and completeness", *Proc. of the 9th International Joint Conference on Artificial Intelligence*, 1985, pp.375-378.

Stachowitz, R.A., and Combs, J.B. [1987a] "Validation of expert systems", *Proc. of the 20th Hawaii International Conference on Systems Sciences*, 1987, pp.686-695.

Stachowitz, R.A., Combs, J.B., and Chang, C.L. [1987b] "Validation of knowledge-based systems", *Proc. of the 2nd AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, Arlington, Virginia, March 9-11, 1987.

Stachowitz, R.A., Chang, C.L., Stock, T., and Combs, J.B. [1987c] "Building validation tools for knowledge-based systems," *Proc. of First Annual Workshop on Space Operations Automation and Robotics,* Houston, Texas, August 1987.

St. Johanser, J.T., and Harbidge, R.M. [1986] "Validating expert systems: Problems & solutions in practice", *Proc. of the International Conference on Knowledge-Based Systems*, London, England, pp.215-229.

Suwa, M., Scott, A.C., and Shortliffe, E.H. [1982] "An approach to verifying completeness and consistency in a rule-based expert system", *The AI Magazine*, 1982, pp.16-21.

Weiss, S.M., and Kulikowski, C.M. [1983] "Testing and evaluating expert systems", Chapter 6 in *A Practical Guide to Designing Expert Systems*, Chapman and Hall, pp.138-156.

Zeil, S.J., and White, L.J. [1980] "Sufficient test sets for path analysis testing strategies", OSU-CISRC-TR-80-6, The Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio 43210, July 1980.

THE AUTHORS

Dr. Chang received his Ph.D. in Electrical Engineering from the University of California, Berkeley, CA in 1967. His background includes design and development of large-scale knowledge-based systems, and research in program generation, very high level languages, compilers, rapid prototyping, relational data bases, natural language query systems, mechanical theorem proving, and pattern recognition. He wrote two books "Symbolic Logic and Mechanical Theorem Proving" (with Dr. Richard Lee), and "Introduction to Artificial Intelligence Techniques", and published more than 50 papers. He is currently a co-principal investigator of the Knowledge-Based Systems Validation project at the Lockheed AI Center.

Dr. Stachowitz received his Ph.D. in Linguistics from the University of Texas at Austin in 1969. His background includes design and development of a large-scale knowledge-based mechanical translation system, computer hardware and software performance evaluation, and research in applicative programming languages, semantic data models, and analytic modeling and performance evaluation of data base machine architectures. He also has performed research in logic and functional knowledge base manipulation and query languages. He is a Senior Research Scientist at Lockheed's Artificial Intelligence Center and co-principal investigator of the Knowledge-Based Systems Validation project.