Shekhar H. Kirani, Imran A. Zualkernan, and Wei-Tek Tsai

Evaluation of **Expert System Testing Methods**

Expert systems are being developed commercially to solve nontraditional problems in such areas as auditing, fault diagnosis, and computer configuration. As expert systems move out from research laboratories to commercial production environments, establishing reliability and robustness have taken on increasing importance [1, 5, 9, 20, 21]. In this article, we would like to assess the comparative effectiveness of testing methods including black-box, white-box, consistency, and completeness testing methods [9, 15–19, 22, 23] in detecting faults.[1]

We take the approach that an expert system life cycle consists of the problem-specification phase[2] [2, 3, 5, 12, 13], solution-specification phase, high-level design phase, implementation phase, and testing phase. This approach is consistent with the modern expert system life cycle as suggested in [9, 21]. *Expert system testing* (generally known as verification and validation [15, 17]) establishes a binary relationship between two by-products of the software-development process. For this article, we consider testing as the comparison between by-products by each life-cycle phase and implementation. We use a technique called "life-cycle mutation testing" (LCMT) for a comparative evaluation of testing methods on an expert system.

Table 1 briefly explains each of the testing methods that are considered in this article [9, 15–17, 19, 23]. Here we comment on characteristics of faults in each life-cycle phase. We make the following general observations about the nature of faults:

- Faults not detected in early phases become progressively more expensive to rectify in later phases.
- A fault-prone region (input region that results in failures) may not be uniformly distributed across the complete input space.
- Faults in early life-cycle phases may induce a multitude of faults in the final program.
- Inconsistency or incompleteness in any phase may result in inconsistent or missing rules in the final program.

Problem specification faults. Problem specification is a description of the problem being solved. Black-box testing methods have been shown to be effective in identifying faults for programs with large fault-prone regions [23]. Therefore, random testing should be effective in detecting problem-specification faults. However, random testing is effective when a fault-prone region is uniformly distributed across the complete input space as opposed to a nonuniform distribution [14] (see Figure 1).

Partition-testing methods are effective when a program has a nonuniform fault-prone region. Performance of input and output partition-testing methods depends on the *partition criteria.* Partition testing methods will perform well if the partition criteria used are consistent with

[1] In this article we adopt the IEEE standard terminology of *error, fault,* and *failure.* According to this terminology errors represent human mistakes that can result in faults in a system. A fault (commonly referred to as a bug) may cause a system to fail on multiple inputs, but each failure can potentially lead to the discovery of a new fault.

[2] For this article we chose to use the term problem specifications as the one that includes a description of the problem, its boundaries, and what needs to be computed (an abstract process description) to solve that problem [21, 25].

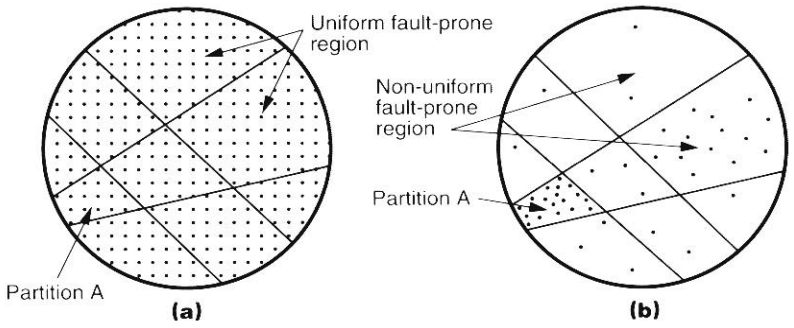

Figure 1. Figure (a) shows partitions in a test space with uniform fault-prone region, and Figure (b) shows partitions in a nonuniform fault-prone region. Each dot represents a fault.

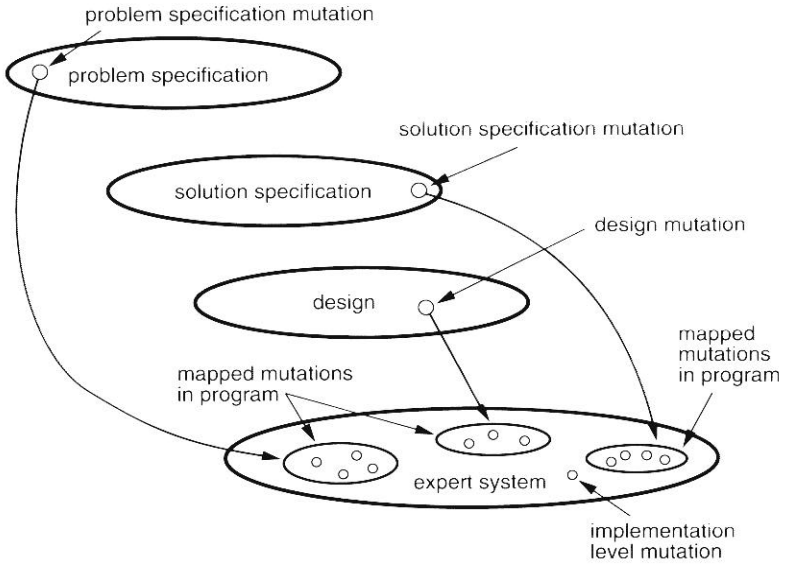

Figure 2. Mutation at various stages of expert system development

how faults in problem specification induce faults in the resulting program. For the situation shown in Figure 1(b) partition criteria are effective if they can identify "Partition A."

Consistency and completeness testing methods are used to identify faults due to redundant, missing, or conflicting rules. Redundant, missing, and conflict rule testing is expected to perform better if faults in problem specification result in redundant, conflicting, and missing rules in the final program.

Solution specification faults. Solution specification describes how to satisfy requirements specified in problem specification. Random testing methods should identify many solution specification faults if these faults result in a program with large uniform fault-prone regions (see Figure 1). As with problem specification, if faults are nonuniformly distributed, partition criteria specifically designed for solution specification faults used in conjunction with partition testing can improve the number of faults that can be caught.

White-box testing methods use the internal structure of the program to generate test cases. Because faults in solution specification affect paths, dynamic-flow testing methods should be effective in catching these faults. Cause-effect testing should also identify many faults in a solution specification, as cause-and-effect relationships are generated from solution specification.

While constructing solution specifications, one may introduce redundant, conflicting specifications or miss some specifications, which may introduce inconsistency and incompleteness in the final program. Redundant, conflict, and missing rule testing should perform better under these circumstances.

Design faults. Design specifies the structure, control, and data-flow information of different modules and the interactions between different modules in the design of a system. If the program has large fault-prone regions, then, as explained earlier, random testing is expected to identify many faults. Input and output partition testing should perform better than random testing if partition criteria uses design information and is designed specifically to identify particular types of bugs.

White-box testing methods (dynamic-flow and cause-effect testing) use design structures (interactions between modules) for testing the program. Because faults in design affect the design structure, white-box testing methods should perform well in identifying such faults.

If design contains redundancy or multiple execution paths that result in conflicting output, or contain missing components, consistency and completeness testing methods are expected to perform well.

Implementation faults. Implementation faults are due to simple mistakes made during implementation, such as initializing with wrong values,

using an incorrect attribute, or typographical mistakes.

We make the following observations about implementation faults:

- Implementation faults result in a program with a small fault-prone region. Because most implementation faults are restricted to single rules, the result is fewer faults in the final program as compared with faults introduced in other phases.
- Faults in implementation affect the execution paths of the program.
- Faults in implementation result in minimal changes in the final program, and therefore, there may not be any redundant, missing, or conflicting rules.

In the implementation phase, black-box testing techniques may not be as effective as in other phases because faults in implementation result in a small fault-prone region. White-box testing techniques use path information, and hence they are expected to perform better than black-box testing techniques, as faults in implementation affect paths. Because faults in implementation result in data-flow anomalies, data-flow testing should be able to identify most implementation faults. Faults in implementation affect many execution paths. This means that dynamic-flow testing should be able to catch most implementation faults.

Implementation faults may result in few inconsistencies in rules like subsumption, redundancy, or conflicting rules. Hence, implementation faults may not be identified by consistency rule checking methods. Illegal attribute value checking and unreferenced attribute value testing should be able to identify implementation faults due to typographical mistakes or wrong value initialization.

Table 1. Different testing methods applicable to expert system.

Name	Description
A) Black-box testing	Test cases are generated without considering how the system solves the problem
Random testing	Test cases are selected randomly from input space
Input partition testing	Input space is partitioned based on a partition criteria, and test cases are picked from partitions
Output partition testing	Test cases are selected from output partitions that are created based on a partition criteria
B) White-box testing	Test cases use the internal structure of a program in addition to input and expected output
Data-flow testing	Analysis of the program for data-flow anomalies related to *define*, *use*, and *kill* of a variable
Dynamic-flow testing	Test cases are generated to exercise different paths of execution in the program
Cause-effect testing	Causes and effects are generated from solution specification, and test cases are constructed by the combinations of causes
C) Consistency testing	Tests the program for internal inconsistencies in rules
Conflict rule testing	Identifies rules that can succeed in the same situation but produce contradictory results
Redundant rule testing	Identifies rules that can succeed in the same situation and produce the same results
Subsumption rule testing	Analyzes program for rules that are subsets of other rules either in conditions portion or in results portion, but not both
D) Completeness testing	Tests the program for internal incompleteness in rules
Missing rule testing	Analyzes the program for identifying missing rules that produce a required result
Unreferenced attribute value testing	Analyzes the program for identifying attributes that are not referenced in any rules
Illegal attribute value testing	Identifies illegal attributes that are used in rules

Case Study Design

To evaluate these propositions, we use LCMT. In mutation testing [7, 10] intentional faults called mutants[3] (or faults) are introduced one at a time into a program, and a testing method to be evaluated is repeatedly applied to the program. If the testing method fails to identify the failures resulting from a mutation, then the mutant is said to be *live*. Otherwise, the mutant is considered *killed*. A testing method's adequacy is determined by the number of mutants it is able to kill. In most previous work on mutation testing, a mutant is introduced only in the program. In LCMT a mutant is introduced in all by-products of development, such as the problem specification, solution specification, high-level design, and the implementation (the program) (see Figure 2).

As an example system we selected a diagnostic VLSI manufacturing expert system called MAPS. We picked MAPS as it is based on *authentic* reasoning methods of an expert with 12 years of experience, and it performed better than an expert with seven years of experience. In addition, since MAPS is small in size, evaluation

[3]In this article we use mutant to specify the actual fault. A mutant type denotes a general class. Mutation and mutant are used interchangeably.

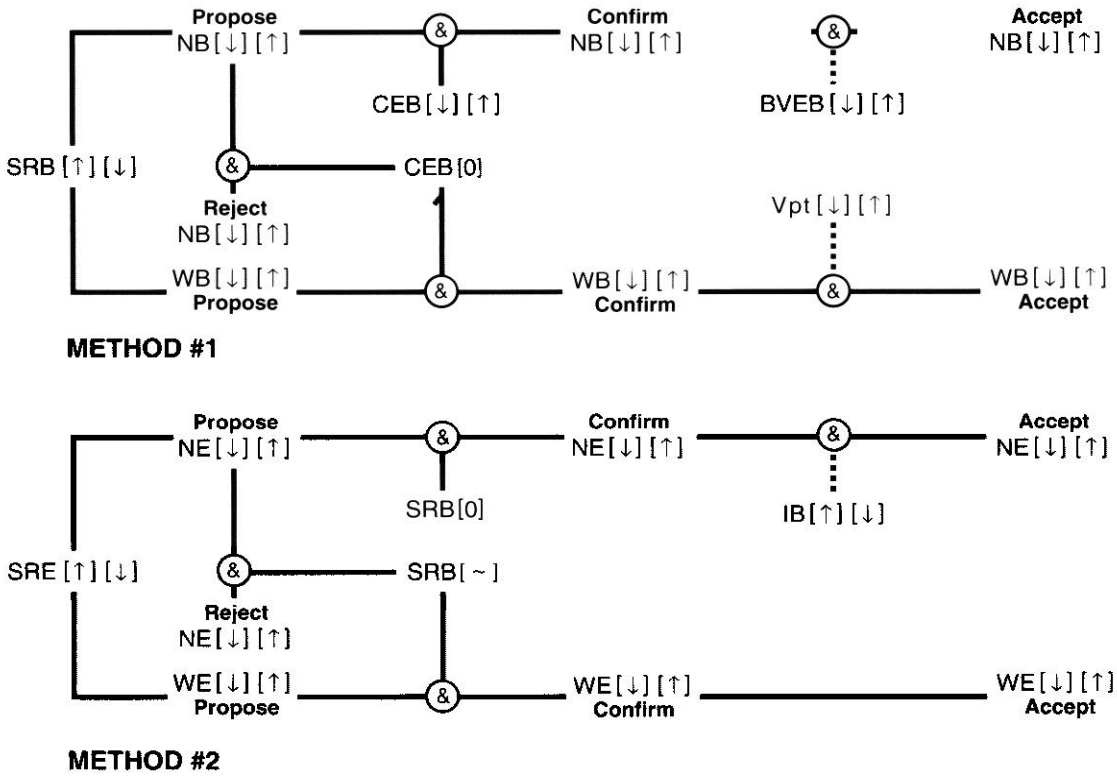

Figure 3. Methods 1 and 2 for solution specification of MAPS

and analysis is easier[11].

Given a set of electrical measurement deviations on a bipolar transistor, MAPS diagnoses a *physical anomaly* that may have caused deviations. An input to MAPS consists of deviations in six electrical parameters, such as *base current* (I_b), *emitter resistance* (SRE), *base resistance* (SRB), *emitter-base junction capacitance* (CEB), *base-collector capacitance* (CBC), and *punch through voltage* (V_{pt}). The output of MAPS consists of physical anomalies. A physical anomaly is a combination of changes in physical parameters, such as widths of the base (W_b), emitter (W_e), collector (W_c), and concentrations of the base (N_b), emitter (N_e), and collector (N_c) of various layers of the transistor. Figure 3 describes the solution specification of MAPS in terms of two methods. Each node in the method represents an operation on a hypothesis or on data. The figure also depicts the way results are proposed, confirmed, and then accepted. The solution specification is derived using protocol analysis of experts [8]. MAPS has a total of 41 rules and is implemented as a forward-chaining production system in common Lisp.

Mutant Types

As a commonly known bug taxonomy does not exist for expert systems, we adapted the *bug taxonomy* for conventional software [4] to construct mutation operators. A mutation operator is a generalized description of a mutant type. For this study, we constructed a total of 90 mutations belonging to each of the four classes of problem specification, solution specification, design, and implementation bugs. Table 2 shows different mutation types introduced in each lifecycle phase of the expert system. For each mutation type we experimented with 10 different mutations in an expert system program. Changes were made in attributes, statements, and rules in the program.

Missing requirement faults are quite common in practice. Two mutation operators consisting of missing input and missing output parameters were constructed in our case study. For feature misunderstood type, mutants were introduced by reversing the direction of deviations of physical anomalies (e.g., width of base). For duplicated logic type of faults, new rules were added such that there was a direct path from *Propose* module to *Accept* module. For unachievable path type of faults, input data at a node were deleted. Similarly, implementation faults were introduced by changing the constant values, typing errors, and exchanging of parameter types in a rule.

Evaluation Scheme for Testing Methods

Testing methods can be classified as *dynamic testing methods* and *static testing methods*. Dynamic testing methods require a test suite (a set of test cases containing test input and output) that must be executed. All black-box and white-box testing methods discussed in this article (see Table 1), except

data-flow testing, are dynamic testing methods. Static testing methods detect faults by analyzing the complete program, but the program is not executed. Data-flow testing, completeness, and consistency testing techniques are static testing methods.

A VLSI process simulator is used for generating the expected output. For dynamic testing, a general test tool was written in Lisp to exercise the program with selected inputs and compare the results with expected output. For partition testing, the tool randomly selected test cases from the partitions (partitions are done based on partition criteria). For dynamic-flow testing, solution specification (see Figure 3) was used in generating test cases. The tool also generated the cause-effect table and cause-effect graph when it was provided with causes, effects, and the program. For data-flow testing, a Prolog-based tool was built that determined *define*, *use*, and *kill* of data values from the program. This data-flow information was analyzed for anomalies. For analyzing the program for consistency and completeness, a table-based iterative tool was built that analyzes the program and determines all the inconsistencies and incompleteness in the rules.

We evaluate testing methods using two measures: *effectiveness* and *robustness*. Effectiveness measures the success of a testing method in identifying faults. Robustness measures the sensitivity of testing methods to the test suite being used. Effectiveness is measured by the number of mutants killed, while robustness is measured by a quantity called *failure ratio*.

Percentage of mutants killed represents a measure of the effectiveness of a testing method. For a given mutant type, percentage of mutants killed is defined as the ratio of number of mutants killed by a testing method to total number of mutants introduced.

For each of the dynamic testing methods, a test suite was derived from a complete enumeration of the input space of MAPS. This test suite consisted of 10% of the total number of randomly selected test cases.[4] MAPS was run on each test case to determine whether it was able to kill the mutant. For each mutant type the number of mutants killed by each dynamic testing method and the percentage of mutants killed were determined.

Static testing methods analyze a complete program and do not entail running of the program. For each mutant type shown in Table 2, 10 mutants were introduced in the program, and the testing method was executed. For each mutant type, the ratio of number of mutants killed by a static testing method to total number of mutations introduced was determined.

Failure ratio indicates the *robustness* of a testing method in killing a mutant. For a given dynamic testing method, percentage of mutants killed, therefore, does not indicate number of test cases in a test suite that killed a mutant. If the test suite had been different, a testing method with a higher failure ratio is more likely to identify a fault than the one with a lower failure ratio.

We define failure ratio as the ratio of number of test cases in a test suite that killed a mutant to total number of test cases in a test suite.

Number of test cases within the test suite that killed the mutant was determined for each of the dynamic testing methods. Failure ratio was determined only for dynamic testing methods as it is not applicable to static testing methods.

[4]We selected 10% of total number of test cases because it is a reasonable representation of total number of test cases. In some expert systems, 10% may represent an extremely large sample size. For example, in MYCIN there are approximately 1×10^{13} possible test cases [6], and 10% of these test cases is too large to use.

Table 2. Mutation (fault) types selected in life-cycle phases. The fault classification is from Beizer [4].

Mutation type	No. of mutations	Total no. of changes in program
A) Problem specification		
Missing requirement Incorrect requirement	10 10	178 66
B) Solution specification		
Feature misunderstood Wrong feature interaction	10 10	52 107
C) Design		
Unachievable path Duplicated logic	10 10	102 55
D) Implementation		
Wrong object access	10	10
Incorrect value initialization Typographical violations	10 10	10 10

Results: Testing Methods Performance

Here we discuss the results of testing methods in killing mutants in all the life-cycle phases. Table 3 shows the percentage mutants killed by different testing methods for each mutation type (see Table 2) in all the life-cycle phases, and Table 4 shows the failure ratio associated with various dynamic testing methods.

From Table 3 it is evident that black-box testing methods killed more mutants in initial life-cycle phases compared with the implementation phase. Table 4 shows that failure ratio for random testing[5] and partition testing is less in implemen-

Table 3. Fraction of mutants killed by testing methods in the software life-cycle phases.

	Software life-cycle phase								
	PS		SS		Design		Implementation		
Testing method	PS-1	PS-2	SS-1	SS-2	DS-1	DS-2	IM-1	IM-2	IM-3
1) Black-box testing									
a. Random	0.6	0.9	1.0	0.5	0.8	0.9	0.7	0.6	0.6
b. Input partition	0.6	0.8	1.0	0.5	0.9	1.0	0.6	0.9	0.3
c. Output partition	0.8	0.8	0.9	0.5	0.9	1.0	0.8	0.6	0.5
2) White-box testing									
a. Data-flow	0.4	0.3	0.1	0.1	0.4	0.6	0.9	0.9	1.0
b. Dynamic-flow	0.7	0.8	0.9	0.8	0.9	1.0	0.9	0.7	0.9
c. Cause-effect	0.7	0.33	0.6	0.53	0.6	0.57	0.6	0.43	0.37
3) Consistency testing									
a. Redundancy rule	0	0	0	0	0	0	0	0	0
b. Conflict rule	0	0	0	0	0	0	0	0.5	0
c. Subsumption rule	0.2	0	0	0	0.2	0	0	0	0
4) Completeness testing									
a. Missing rule	0.4	0.3	0.1	0	0.6	0	0.3	0	0
b. Unreferred attribute	0.6	0.4	0.3	0.2	0.4	0.6	0.3	0	0.1
c. Illegal attribute	0	0.4	0	0	0	0	0.2	0.5	0.5

PS: problem specification and SS: solution specification;
PS-1: missing requirement fault and PS-2: incorrect requirement;
SS-1: feature misunderstood fault and SS-2: wrong feature interaction fault;
DS-1: unachievable path fault and DS-2: duplicated logic fault;
IM-1: object acess fault; IM-2: value initialization fault; IM-3: typographical fault.

Among the white-box testing methods, dynamic-flow testing was effective throughout the life-cycle and identified maximum number of mutants in the design phase. Data-flow testing killed more mutants in all the mutant types of implementation. Figure 4 shows that average failure ratio for dynamic-flow testing was the maximum across the life-cycle phases. Average failure ratios for all other testing methods were similar. Thus

[5]It is essential to note that in this case study, expected outputs are generated without much cost by running a simulator. In case such a simulator is not available, it would be extremely difficult to carry out random testing.

As shown in Table 3, all consistency testing methods performed poorly compared with other testing methods. Redundant rule testing did not kill any mutants in any life-cycle phases. Data-flow testing found more failures in implementation phases as compared with other phases. As Table 3 shows, completeness testing methods performance was better than consistency testing methods. Table 3 shows that missing rule testing performed best in design phases as compared with other phases. Few faults were recorded for unreferenced attribute testing in implementation. Illegal attributes testing found more mutants in implementation and problem specification compared with other testing methods.

Figure 5 shows percentage of mutants killed by testing methods in each phase of life cycle. Results are shown by combining the individual results of mutation types in each phase. Black-box testing and white-box testing methods performed well throughout the life cycle compared with consistency and completeness testing methods.

Discussion

In this section we discuss the performance of testing methods with respect to propositions made earlier. The assumptions that the testing methods confirmed are the following:

1. Faults in initial life-cycle phases resulted in a multitude of faults in the final program. This assumption was confirmed in this case study (see Figure 6).
2. Faults in problem specification, solution specification, and design resulted in missing rules and unreferenced attributes in the final program. This is because mutations in earlier phases of life-cycle affected a large number of rules in final program. Whenever rules got deleted it resulted in many unreferenced attributes.

The testing methods that confirmed our expectations are:

1. Black-box testing methods (random, input-partition, and output-partition) were effective throughout the life-cycle phases (see Figure 5). From Table 2 we see that faults in problem specification, solution specification, and design result in a multitude of faults in the program. However, faults in the implementation phase do not induce faults, and faults are restricted to a single rule. Failure ratio for black-box testing methods were high in problem specification, solution specification, and design compared with implementation. Thus black-box testing methods performed well in all phases except in implementation.
2. Dynamic-flow testing (white-box) was effective and robust throughout the life-cycle phases and was most

effective in the design phase (see Table 3). In design phase modules and the data flow, control flow and the interactions between modules are specified. Dynamic-flow testing uses the design components and the paths of execution between components for generating test cases. Because faults in design are related to paths, dynamic-flow testing was successful in identifying most of the mutants. In addition, dynamic-flow testing had a maximum failure ratio in all life-cycle phases.

3. Data-flow testing killed maximum number of mutants in implementation phase as compared with other phases.

In implementation phase, a common fault is to compare the value of an attribute without defining it. A typographical mistake in use of an attribute in a rule may result in a data-flow anomaly. Hence, data-flow testing is able to catch more faults in the implementation phase compared with other phases. Mutations in earlier phases of life cycle usually modified or removed complete rules, and this may not result in many data-flow anomalies.

4. Illegal attribute testing identifies more faults in implementation compared with other life-cycle phases.

Table 4. Failure ratio of dynamic testing methods in all the software life-cycle phases.

Testing method	Software life-cycle phase								
	PS		SS		Design		Implementation		
	PS-1	PS-2	SS-1	SS-2	DS-1	DS-2	IM-1	IM-2	IM-3
1) Black-box testing									
a. Random	0.18	0.3	0.33	0.24	0.38	0.32	0.15	0.12	0.1
b. Input partition	0.18	0.26	0.35	0.28	0.39	0.34	0.21	0.13	0.4
c. Output partition	0.18	0.25	0.33	0.31	0.39	0.27	0.18	0.10	0.12
2) White-box testing									
a. Dynamic-flow	0.32	0.24	0.39	0.47	0.44	0.45	0.35	0.20	0.21
b. Cause-effect	0.25	0.13	0.32	0.27	0.29	0.19	0.22	0.11	0.8

PS: problem specification and SS: solution specification;
PS-1: missing requirement fault and PS-2: incorrect requirement;
SS-1: feature misunderstood fault and SS-2: wrong feature interaction fault;
DS-1: unachievable path fault and DS-2: duplicated logic fault;
IM-1: object access fault; IM-2: value initialization fault; IM-3: typographical fault.

Illegal attribute testing only killed mutants in the implementation phase. This is because most of the illegal attributes are due to typographical errors.

The testing methods whose performances were different than expected are:

1. Consistency testing methods (redundancy, conflict, and subsumption rule testing) performed poorly throughout the life-cycle phases (see Table 3). Redundant and conflicting specifications in the initial life-cycle phase did not propagate to the final program.

Redundancy and conflict in a life-cycle phase affects several rules, and all rules taken together may result in redundancy or conflict. In redundancy and conflict rule testing, rules are checked pair-wise for redundancy and conflict.

2. Subsumption rules in final programs may result even if there are no subsumptions in earlier phases of the life cycle. Subsumption rule testing was not effective in killing mutants (see Table 3). Mutations did not result in subsumptions in any life-cycle phase. However, few subsumption rules were identified in the final program while identifying mutants in problem and solution specification phases.

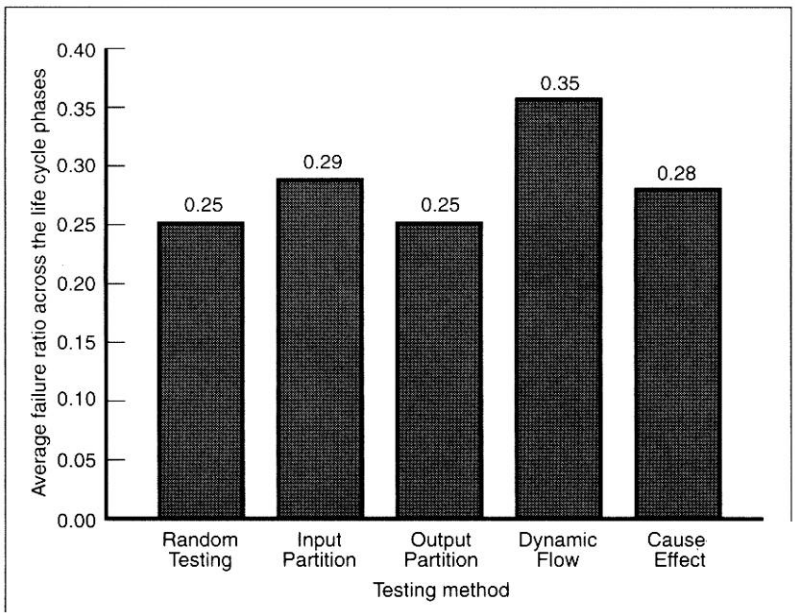

Figure 4. Average failure rated across the life-cycle phases

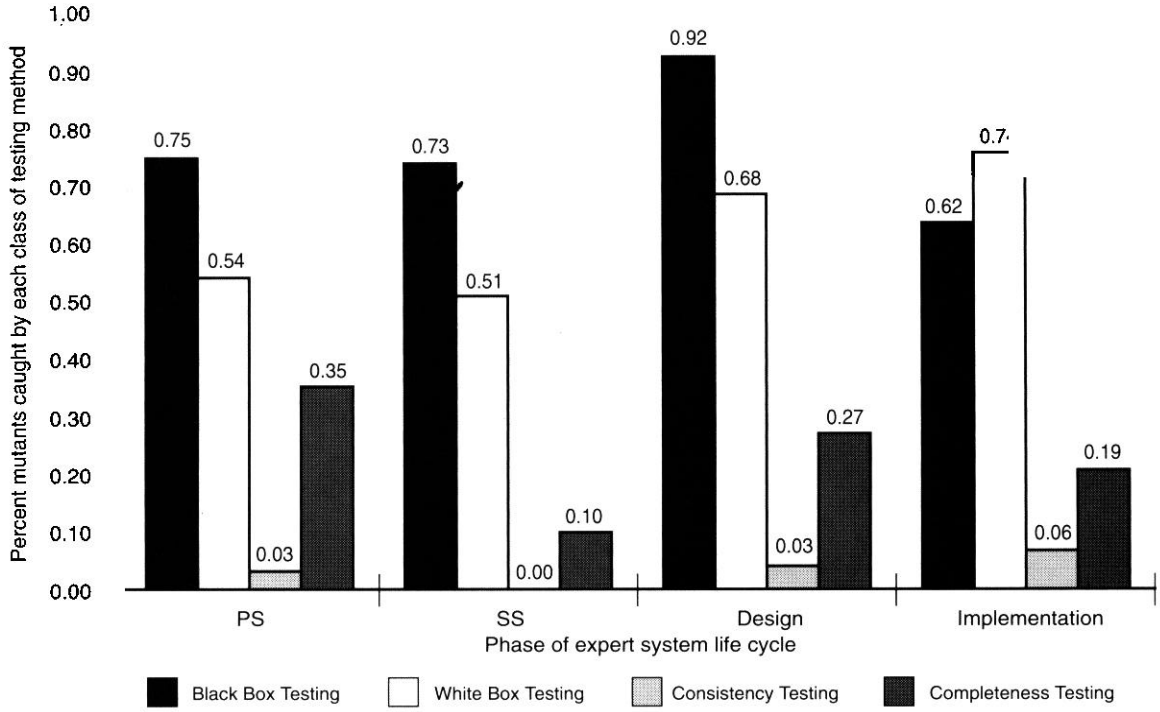

Figure 5. Percentage of mutants killed by black-box, white-box, consistency, and completeness testing methods in life-cycle phases. (PS: problem specification and SS: solution specification)

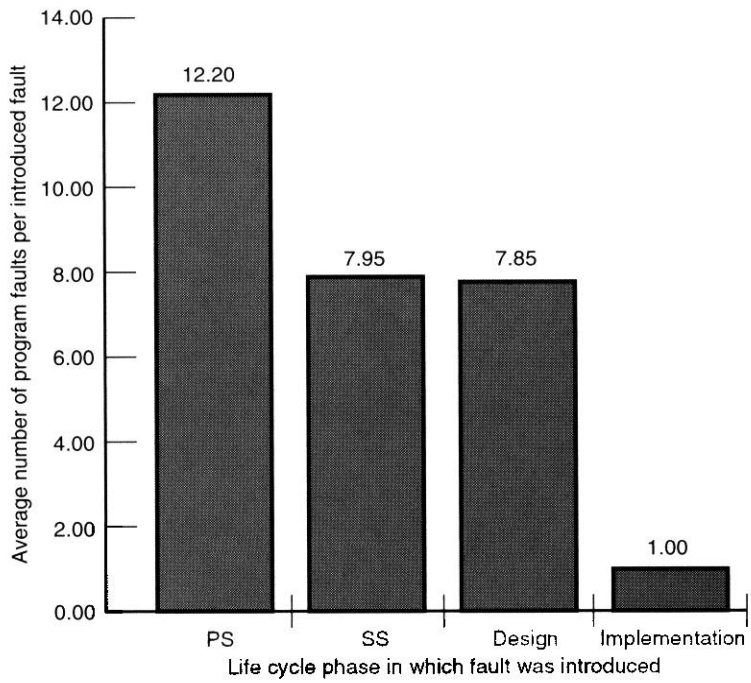

Figure 6. Number of faults in the program due to faults in life-cycle phases

Cost Associated with Testing Methods

Each testing method performed differently in identifying faults in various expert system life-cycle phases. In this section we discuss the cost-related issues with each testing method. Total cost associated with a testing method depends on cost of test case generation, cost of evaluation, and cost of loss.

Cost of test case generation consists of cost associated with generation of test inputs and expected outputs. The test inputs are determined using the techniques specified by various testing methods. Generating expected outputs is often difficult because testing criteria can be unclear or due to lack of oracles [23]. Consistency and completeness testing methods have less cost of generation compared with dynamic testing methods

(see Table 5).

We can assume that the cost associated with test case evaluation is generally less compared with the cost of test case generation as it involves only computer time. For consistency and completeness testing methods cost associated with running a test program on rule base is less than dynamic testing methods.

Cost of loss is the cost incurred if a testing method fails to identify a fault. Thus the cost of loss depends on the effectiveness of a testing method in catching faults. Cost of missing faults depends on the life-cycle phase in which a testing method is applied. Table 6 shows testing methods ordered in increasing sequence of cost of missing faults in each life-cycle phase. Percentage of mutants killed by a testing method is used as the measure to order the cost.

We can compute the cost of loss of testing methods over all life-cycle phases as follows:

$$Cost^{m}_{loss} = \sum_{i} (1 - \mu^{m}_{i}) * \omega_{i},$$

where

$\sum_{i} \{PS, SS, Design, and\ Implementation\}$,
μ^{m}_{i} : percentage of mutants killed by a testing method m in phase i,
ω_{i} : relative weights of cost of missing a fault in phase i.

Table 7 shows ordering of testing methods based on their $Cost_{loss}$ over all life-cycle phases. The weights correspond to multiplication factors for missing a fault in a life-cycle phase [24]. From Table 7 we see that $cost_{loss}$ has remained the same for different weight vectors. Dynamic-flow testing and output-partition testing has the least $cost_{loss}$ while conflict rule testing and redundant rule testing have the maximum $cost_{loss}$. Further studies must be done to determine actual weight sequences for expert systems to compute the actual $cost_{loss}$.

In real-life testing, one would prefer testing methods that are cost-effective and are able to identify most of the faults across the life-cycle phases. Considering cost and perfor-

Table 5. Testing methods ordered in increasing order of costs from top to bottom

Cost of test case generation	Cost of test case evaluation
Consistency and completeness testing methods	Illegal attribute
Data-flow	Unreferenced attribute
Random	Missing rule
Input partition	Redundancy rule
Output partition	Conflict rule
Dynamic-flow	Subsumption rule
Cause-effect	Data-flow
	Random
	Input partition
	Output partition
	Cause-effect
	Dynamic-flow

Table 6. Testing methods ordered in increasing order of cost of missing faults from top to bottom.

PS	SS	Design	Implementation
Random	Dynamic-flow	Dynamic-flow	Data-flow
Dynamic-flow	Random	Partition	Dynamic-flow
Partition	Partition	Random	Random
Cause-effect	Cause-effect	Cause-effect	Partition
Unreferenced attribute	Unreferenced attribute	Unreferenced attribute	Cause-effect
Data-flow	Data-flow	Data-flow	Illegal
Missing rule	Missing rule	Missing	Unreferenced attribute
Illegal attribute	Conflict	Subsumption	Conflict
Subsumption	Redundancy	Conflict	Missing
Conflict	Subsumption	Redundancy	Subsumption
Redundancy	Illegal attribute	Illegal attribute	Redundancy

PS: problem specification and SS: solution specification.

Table 7. Testing methods ordered in increasing order of cost of loss for different weight sequence.

Weights (ω) (PS, SS, Des, Imp)	Testing methods ordered in increasing $cost_{loss}$
(28.8, 4, 2, 1) (32.4, 4.5, 2.25, 1) (36, 5, 2.5, 1) (39.6, 5.5, 2.75, 1) (43.2, 6, 3, 1)	(Dynamic-flow, Output-partition, Random, Input-partition, Cause-effect, Unreferenced attribute, Data-flow, Missing, Illegal attribute, Subsumption, Conflict, Redundant)

PS: problem specification, SS: solution specification, Des: design, Imp: implementation phase. The weight vector (PS, SS, Des, Imp) corresponds to multiplication factors for missing a fault in corresponding life-cycle phase.

mance together, the cost-effective testing methods were:

1. Random and partition testing methods in problem specification phase.
2. Partition and dynamic-flow testing methods in solution specification phase.
3. Unreferenced attribute testing and partition testing in design phase.
4. Data-flow testing, illegal attribute testing, and black-box testing methods in implementation phase.
5. Partition testing and dynamic-flow testing methods if we consider all the life-cycle phases.

Conclusion

In this article we have demonstrated a new technique—life-cycle mutation testing—for evaluating different testing methods for expert systems. We have demonstrated the use of this technique for a VLSI diagnostic expert system on black-box, white-box, consistency, and completeness testing methods.

The results of our study are that dynamic-flow testing can be used for catching faults in all the phases of software development. Random testing, input-partition testing, and output-partition testing were effective throughout the life cycle of expert systems. Data-flow testing was effective in identifying faults in implementation. Conflict, redundancy, and subsumption testing were not common in expert systems. Missing rule testing and unreferenced attribute testing can be used for catching faults in initial stages of software development. Illegal attribute testing was most effective in the implementation phase. Considering the cost associated with testing, dynamic testing and output-partition testing were the cost-effective testing methods for all the life-cycle phases.

The results are clearly limited to one system and a particular set of fault types. The testing techniques are demonstrated for forward chaining systems. For backward chaining systems, black-box testing methods remain the same, but differ with respect to white-box testing in the way paths are generated. Extensive evaluation of various additional systems (both forward and backward chaining systems) with a more extensive bug classification is necessary to derive the relationships between different faults, testing methods, and life-cycle phases. However, method of evaluation of testing methods will be the same as shown in this article. ◻

Acknowledgments

We would like to thank C. Caswell and her colleagues at IBM, East Fishkill, N.Y., for their contribution to the development of MAPS, the example expert system used in this article.

References

1. Barker, V.E. and O'Connor, D.E. Expert systems for configuration at digital: XCON and beyond. *Commun. ACM 32*, 3 (1989), 298–318.
2. Batarekh, A., Preece, A.D., Bennett, A., and Grogono, P. Specification of expert systems. In *Proceedings of IEEE International Conference on Tools for AI*, 1990, pp. 103–109.
3. Batarekh, A., Preece, A.D., Bennett, A., and Grogono, P. Specifying an expert system. *Expert Systems With Applications* 2 (1991), 285–303.
4. Beizer, B. *Software Testing Techniques*. 2d ed. Van Nostrand Reinhold, New York, 1990.
5. Bologna, S., Ness, E., and Sivertsen, T. Dependable knowledge-based systems development and verification: what we can learn from software engineering and what we need. In *Proceedings of IEEE International Conference on Tools for AI*, 1990, pp. 86–95.
6. Buchanan, B.G. AI as an experimental science. In *Aspects of Artificial Intelligence*, J.E. Fetzer, Ed. Kluwer Academic Publishing, Norwell, Mass., 1988, pp. 209–250.
7. DeMillo, R.A., McCracken, W.M., Martin, R.J., and Passafiume, J.F. *Software Testing and Evaluation*. The Benjamin-Cummings Publishing Company, Inc., Redwood City, Calif., 1987.
8. Ericsson, K.A., and Simon, H.A. *Protocol Analysis*. MIT Press, Cambridge, Mass., 1984.
9. Gupta, U.G. *Validating and Verifying Knowledge-based Systems*. IEEE Computer Society Press, Los Alamitos, Calif., 1991.
10. Howden, W.E. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering, SE-8*, 4 (July 1982), 371–379.
11. Johnson, P.E., Kochevar, L., and Zualkernan, I.A. Expertise and fit: aspects of cognition. In *Proceedings of the Symposium on Study of Cognition: Conceptual and Methodological Issues* (Minneapolis, Minn., Feb.–Mar. 1991).
12. Jonker, W., Spee, J., Veld, L., and Koopman, M. Formal approaches toward design in SE and their role in KBS design. In *IJCAI-91 Workshop on Software Enginnering for Knowledge-Based Systems*, August 1991, pp. 84–98.
13. Liebowitz, J., and De Salvo, D.A., Eds. *Structuring Expert Systems: Domain, Design and Development*. Yourdon Press, Prentice Hall, Englewood Cliffs, N.J., 1989.
14. Loo, P.S., Tsai, W.T., and Tsai, W.K. Random testing revisted. *Information and Software Technology 30*, 7 (1989), 402–417.
15. Myers, G. *The Art of Software Testing*. Prentice Hall, Englewood Cliffs, N.J., 1979.
16. Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D. Knowledge base verification. *AI Mag.* (Summer 1987), 65–79.
17. Ould, M.A., and Unwin, C. *Testing in Software Development*. Cambridge University Press, New York, 1987.

18. Partridge, D. *Artificial Intelligence: Applications in the Future of Software Engineering*. Addison-Wesley, Reading, Mass., 1986.
19. Suwa, M., Scott, A.C., and Shortliffe, E.H. An approach to verifying completeness and consistency in a rule-based expert system. *AI Mag.* (Fall 1982), 16–21.
20. Tsai, W.T., Heisler, K., Volovik, D., and Zualkernan, I.A. A critical look at the relationship between AI and software engineering. In *Proceedings of the IEEE Workshop on Languages for Automation*, 1988, pp. 2–18.
21. Tsai, W.T., Heisler, K., Volovik, D., and Zualkernan, I.A. An analysis of expert system life cycle. In *Proceedings of International Conference on New Generation Computer Systems*, 1989, pp. 389–405.
22. Tsai, W.T. and Zualkernan, I.A. Towards a unified framework for testing expert systems. In *Proceedings of Software Engineering and Knowledge Engineering*, 1990.
23. Tsai, W.T., Zualkernan, I.A., and Kirani, S. Pragmatic testing methods for expert systems. *COMPAC-92* (1992), 320–335.
24. Wolverton, R.W. Software costing. In *Handbook of Software Engineering*, C.R. Vick and C.V. Ramamoorthy, Eds. Van Nostrand Reinhold, New York, 1984, pp. 469–493.
25. Zualkernan, I.A., and Tsai, W.T. Are knowledge representation the answer to requirement analysis? In *Proceedings of IEEE Computer Languages*, 1988, pp. 437–443.

About the Authors:

SHEKHAR KIRANI is a member of technical staff at US WEST Technologies. He is also a Ph.D. candidate in computer science at the University of Minnesota. Current research interests include software engineering and object-oriented systems. **Author's Present Address:** US WEST Technologies, 4001 Discovery Drive, Suite 270, Boulder, CO 80303, kirani@advtech.uswest.com

IMRAN ZUALKERNAN is an assistant professor of computer engineering at Penn State University. Current research interests include software engineering of expert systems and knowledge based software engineering. **Author's Present Address:** Dept. of Electrical and Computer Engineering, Penn State University, University Park, PA 16802, zualkern@paris.ece.edu

WEI-TEK TSAI is an associate professor of computer science at the University of Minnesota. Current research interests include software engineering, computer security, and parallel processing. **Author's Present Address:** Computer Science Dept., 200 Union St., Minneapolis, MN 55455, tsai@cs.umn.edu

This work was supported by the National Science Foundation under Grant IRI-891998501.