Automatic Tools for Testing Expert Systems

Uma G. Gupta

In the last several years, validating and verifying (V&V) expert systems has come a long way with a steady increase in the number of articles that emphasize the importance of reliable and rigorous test tools for the success of expert systems [5]. There are several reasons why V&V is critical for intelligent systems. First, mission-critical applications demand unconditional performance guarantees. Second, as systems become larger and more complex, the challenges of testing such systems increases exponentially. Third, the threat of law suits from malfunctioning systems is a serious concern for system developers. Fourth, expert systems may take anywhere from 6 workyears to 15 to 30 workyears to develop and their operational success depends on reliable test techniques. Fifth, lack of rigorous V&V tools continues to be a primary reason for the few documented successful systems [6].

Validation refers to evaluating system performance to establish compliance with functional requirements and assess system accuracy and correctness by addressing modeling questions such as have we built the right system?; is the system knowledge adequate? Domain validation ensures accuracy and completeness of the knowledge base while procedural validation establishes the accuracy and reliability of system output [3]. Verification, on the other hand, establishes structural correctness and process effectiveness by testing the logic of the knowledge base while testing executes a piece of software with the goal of finding errors; structural testing exercises a set of test cases on as many paths as possible, although it does not guarantee that each path is tested while functional testing validates problem specifications by comparing system output with known results. Both functional and structural testing are necessary to build reliable systems[1].

Current tools and techniques for testing expert systems include test case generation, face validation, Turing test, field tests, subsystem validation, and sensitivity analysis. Since there is no one single test technique that captures all errors, developers must try a combination of different methods [2].

Of these techniques, test case generation continues to be the most popular technique. Some limitations of manual test case generation can be overcome by automatic test case generators: software that automatically identifies a set of input-output pairs, where the input identifies the path(s), conditions, and condition values to be tested while the output identifies the results associated with the input. Other approaches and tools to automating the testing process include Expert Systems Validation Associate (EVA) [4], dependency charts, decision tables, graphs or Petri nets, and exploration of dynamic and temporal relationship between rules [8, 9].

Overview of RITCaG

RITCaG is an automatic, object-oriented test case generator that hierarchically tests the performance of a rule-based expert system. It is coded in Symbolics Lisp and was developed on ART (Automated Reasoning Tool), an expert system shell built by Inference Corp.
RITCaG has five primary objectives.

- Generic unique and relevant test cases to validate system performance
- Facilitate incremental testing of a knowledge base
- Validate system performance and system brittleness in a flexible, user-controlled, test environment
- Allocate test resources based on user preference
- Maintain a history of system testing

RITCaG has five modules, including:

Context Builder. Software partitions facilitate code reusability, improve software maintenance, enhance test rigor, and result in robust and reliable systems. The idea of partitioning AI software was first conceived as "planning islands" by Marvin Minsky who argued that partitions greatly reduce the search space, particularly for combinatorial problems. Currently, there are several partitioning strategies such as dividing a large knowledge base into smaller ones, grouping rules using key words, using data dependencies between rules and their informational content, implementing disjointed sets, clustering analysis, separating the knowledge base from the working memory, graph theory, information-theoretic partitioning approaches and hierarchical layering. However, because of the complexity of hybrid partitioning strategies and the lack of clear guidelines on *how* to partition and *how much* to partition, many developers are reluctant to use these strategies.

RITCaG uses three strategies to interactively partition the knowledge base into contexts where a context is a logical grouping of rules: common goals(s) shared by different rules, domain characteristics, and control and logic constraints. RITCaG automatically generates a tabular representation of all contexts in the knowledge base in the form of a scrollable zero-one matrix, referred to as the *Context Matrix* and a scrollable, zero-one matrix showing the antecedents and consequence of each rule in a sures the priority of a rule relative to other rules in the knowledge base, and condition priority, a measure of the number of times a given condition appears in the knowledge base.

Context Analyzer. This module analyzes the contents of each context, rule, and condition using object-oriented methodologies and equivalence classes. There are six object classes: Context Class, Rule Class, Condition Class, Context Test Case Class, Rule Test Case Class, and Condition Test Case Class (see Figure 1). The classes and their instances provide the framework for generating test cases.

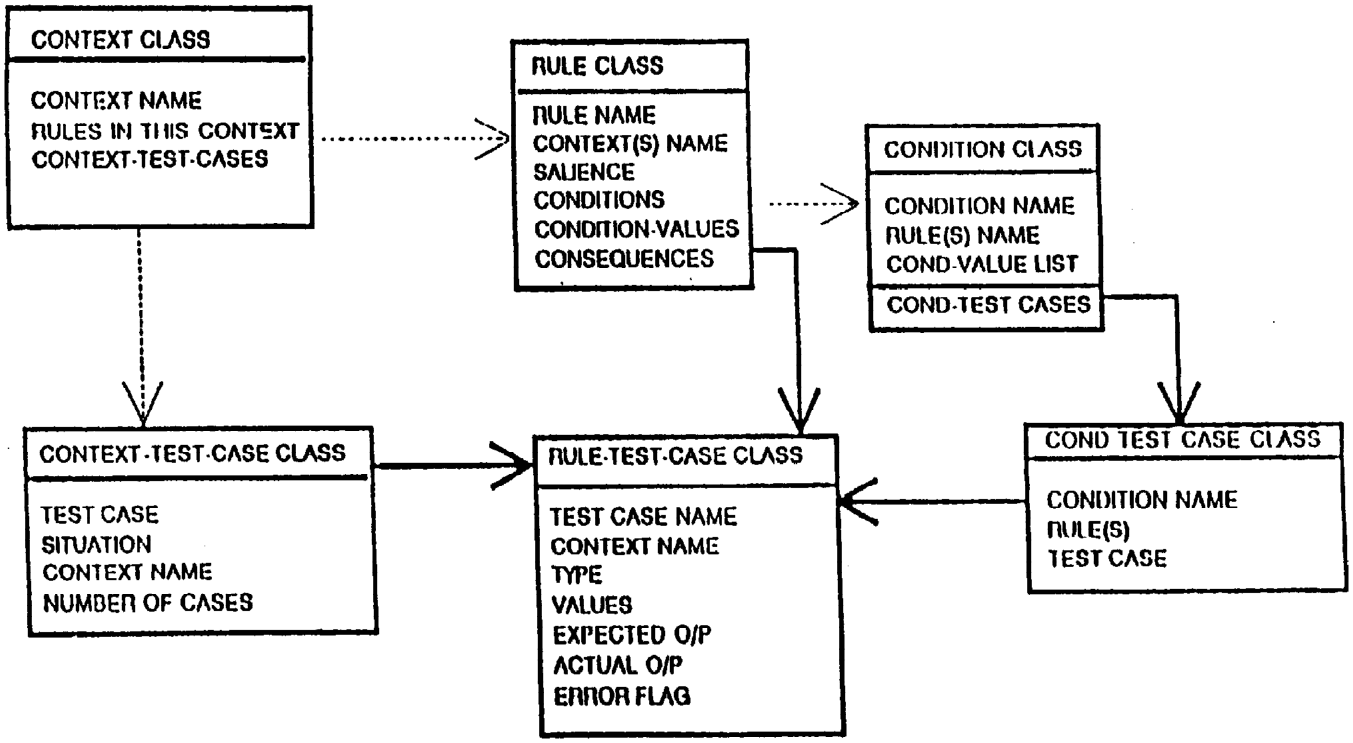

Figure 1

Equivalence classes partition the input domain into a finite number of classes where each member in a given class has an equal probability of detecting system errors, thus allowing testers to derive the benefits of exhaustive testing using a much smaller test. The number of values in each equivalence class depends on factors such as domain characteristics, criticality of test values, and condition values [8].

This module creates two types of equivalence classes for each condition in the knowledge base: *Legal Equivalence Class (LEC)* and *Illegal Equivalence Class (ILLEC).* LEC tests functional specifications by validating system performance in the range of legal inputs while the ILLEC tests a system's brittleness in the range of illegal inputs. In both cases, boundary values are included to push the limits of the system for scope and error recovery. The number of values in the LEC depends on the number and the data type of each condition while the total number of unique knowledge units is a product of the number of values for each rule condition. Hence, as the size of the LEC increases, the number of unique knowledge units increases exponentially.

Knowledge Unit Generator. The Knowledge Unit Generator generates unique test cases, where a unique test case consists of a set of knowledge units and its associated output. A case is unique if its knowledge units are different from other test cases associated with the given rule, where a *knowledge unit* is the smallest unit of knowledge consisting of a condition and its value. Creating an exhaustive set of unique knowledge units for each rule is a one time process and the search space is reduced at the end of each test cycle by removing the knowledge units already generated.

The Knowledge Unit Generator has four important functions:

- Create LEC and ILLEC for each condition in a given rule.
- Use the LEC to identify an exhaustive set of unique knowledge units associated with a given rule.
- Randomly select a condition value from the above set.
- Reduce the search space by eliminating the selected set from the search space.

For example, suppose we have the following rule:

IF

Rotation >= 10
Temperature = High or Medium
Gauge <-5 OR >= 20

THEN

Mechanical-Trouble='Likely'

The values in the LEC set are Rotation (5.01, 10), Temperature ('High', 'Medium'), and Gauge (-5.01, 20, 21) and these values are used in generating Error-free test cases. Figure 2 shows how unique knowledge units are generated by KUG. Error-seed test cases designed to test system brittleness are generated by randomly selecting and assigning values from the ILLEC set to any *one* condition in a rule, while other conditions are assigned legal values.

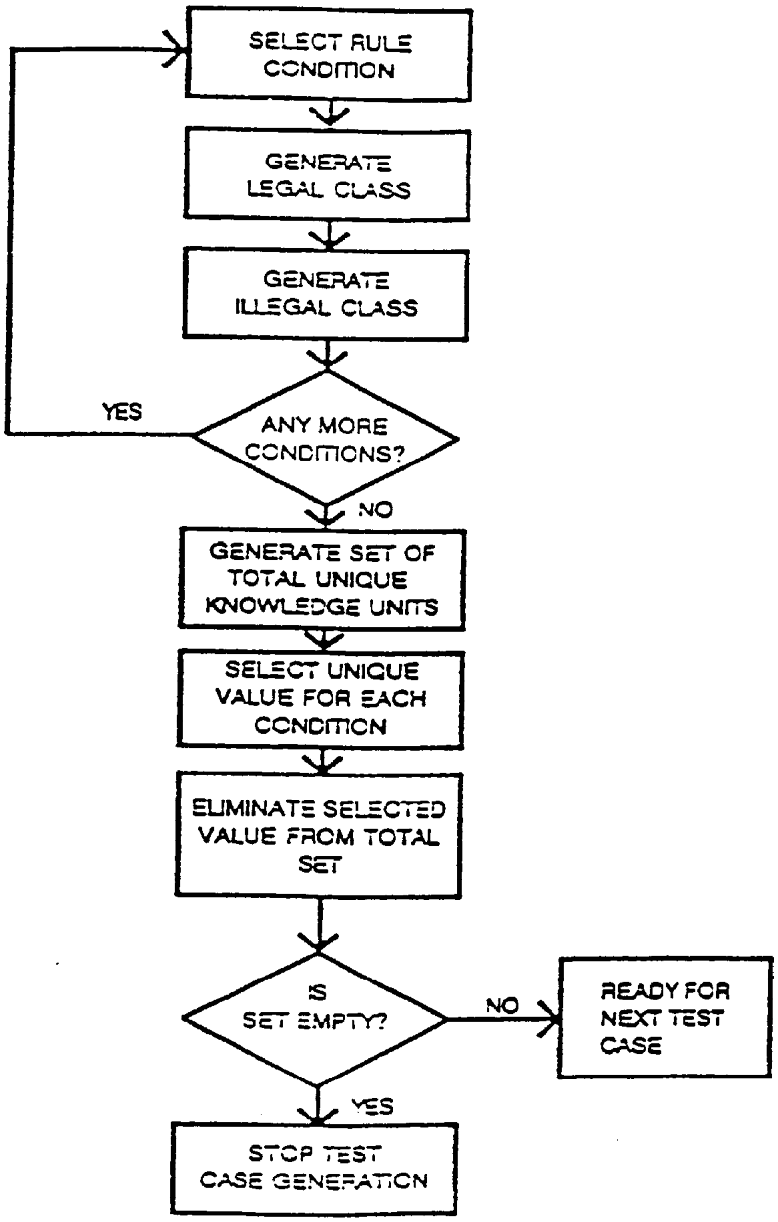
SELECT RULE CONDITION
GENERATE LEGAL CLASS
GENERATE ILLEGAL CLASS
ANY MORE CONDITIONS?
YES
NO
GENERATE SET OF TOTAL UNIQUE KNOWLEDGE UNITS
SELECT UNIQUE VALUE FOR EACH CONDITION
ELIMINATE SELECTED VALUE FROM TOTAL SET
IS SET EMPTY?
NO
READY FOR NEXT TEST CASE
YES
STOP TEST CASE GENERATION

Figure 2

Hierarchical View of Testing an Expert System

A hierarchical view of a knowledge base has three levels: contexts, rules, and conditions. Conditions form the foundation of the knowledge base (Level 3), while a set of conditions, i.e., a rule, is Level 2 in the hierarchy, and a set of rules is a context or partition which is Level 1. These three levels are tested using three approaches: Context-wise Test Case Generation (CX-TC) (to test context performance), Rule-wise Test Case Generation (RU-TC) (to test rule performance), and Condition-wise Test Case Generation (CO-TC) (to test the performance of a given condition).

Context-wise (CW) testing is similar to unit testing in conventional software where a given context (or all contexts triggered by the activation of a given context) is tested using a set of Error-free test cases; a set consists of one Error-free test case for each rule in the context.

Rule-Wise (RW) testing tests a single rule or group of rules by exploiting rule independence to achieve partial modularity, although unexpected or overlooked interactions between rules is a limitation. In the case of a single rule, a permutations algorithm calculates the total number of unique test cases (T) associated with the rule and the Knowledge Unit Generator generates a total of M Error-free and Error-seed test cases, where M T. The number of Error-free and Error-seed test cases is determined using seed-level (S), which is the number of Error-seed test cases as a percentage of total number of test cases (M). KUG generates M * S *Error-seed* test cases and (M * (1 - S)) *Error-free* test cases and each test case becomes an instance of the Rule-Test-Case class. This approach allows developers to validate and analyze the impact of additions and deletions, improve error-correction and error-detection at a micro level, minimize the impact of unexpected or undesirable rule interactions, and distribute valuable test resources between testing system performance and system brittleness, based on domain characteristics and nature of domain problems.

Condition-wise (CN) analyzes the performance of all rules that use a given condition. Using three parameters, namely, number of test cases to generate (M) (user-determined), seed-level (S) for *each* rule that uses the given condition., and number of unique test cases (T), it generates the appropriate number of *Error-free* and *Error-seed* test cases and stores them as instances of the Condition Test Case Class. The module automatically identifies the rules that are affected by a system change.

The Activator executes a sub set of test cases generated by the previous module, analyzes the outcome of each test case, and creates an error file. The test cases can be selected by the developer or the system can do this automatically. The test cases are inserted into ARTs data base which then activates the three-step reasoning cycle: *match* (match date in the database with rules in the knowledge base), *select (select and activate the matching rules)*, and *fire* (activate the rule with the highest salience). It then updates the data base and the reasoning cycle begins all over again.

RITCaG tests for external validity by comparing system performance with known or expected results (consequent of the last rule that was fired or expert opinion). Successful test cases, i.e., those that detect an error, are stored in an error file. If no errors are found, the system checks to see if the fired rule is a part of a *situation*, where a situation is defined as a sequence of dependent rules (a rule that is triggered by an active rule) that may span different contexts. There are two types of situations: *tightly-coupled*, an ordered set of dependent rules triggered in sequence, *and loosely-coupled situations*, a set of rules that share a common antecedent. Thus, a *situation* exemplifies the notion of an execution path in an expert system, where a path may be viewed as one or more chains of inter-dependent rule firings, with the primary goal of capturing as many paths as possible. Note that a rule that activates more than one rule results in multiple situations.

Situations are detected dynamically using ARTs *watch* feature. When a rule is fired the *watch* mechanism dynamically identifies the rules instantiated by the fired rule and by comparing the agenda *before* and *after* a rule was fired, the system automatically and dynamically detects tightly-coupled situations. *Loosely-coupled situation* are used simply to detect the source of the error.

Once an error is detected, the Activator records the test case in an error file and the system executes the next test case. If there is no error and if the last fired rule is not part of a situation, the agenda will be empty. But if the fired rule is part of a situation, the agenda displays rules triggered by the fired rule and the Knowledge Unit Generator generates a unique test case for each rule on the agenda (i.e., rules that make up the situation). If the system detects an error while testing a situation, it generates an error message, records the inference path where the error was found, and abandons further testing. The only exception to the above process of testing situations is in Context-wise testing where RITCaG allows the tester to determine if a situation should be tested or not, independent of error-detection.

History Tabulator. The History Tabulator maintains a complete and comprehensive history of all executed test cases, thus allowing the tester to identify and measure the scope and extent of testing. It has four important functions:

- Maintain a history of all test cases generated by the Knowledge Unit Generator.
- Maintain a history of all test cases executed on the system.
- Identify each successful test case in an error file.
- Maintain a record of all contexts, rules, and conditions, in the knowledge base that are *not* yet tested.

Validation of RITCaG

There are four types of errors in rule-based systems: domain errors, computation errors, coincidental correctness, and missing path errors [9]. RITCaG was validated for domain errors and computational errors; coincidental correctness and missing path errors are beyond the scope of this research. Two small expert systems (less than 25 rules) were used to validate RITCaG and the errors detected in the first system were used to correct and validate the second system. Since it was difficult to locate an operational system that was developed on ART, testing has been limited to demos.

Future Research

Three areas are prime for future research: First, there is a need for more robust and reliable partitioning algorithms; second, validating the performance of uncertainty factors in intelligent systems is still in its infancy; third, hybrid knowledge representation schemes are common in many operational systems and automatic test tools for testing such systems are needed. RITCaG can be enhanced to generate test cases with unique knowledge units and unique test goals. Currently, a user selects test cases that he or she perceives as having unique test goals. However, a path traversal algorithm that automatically discards test cases that traverse the same path will be very useful.

References

1.Adrion, W. R., Branstad, M. A., Cherniavsky, J. C. Validation, verification, and testing of computer software. *Comput. Surv 14*, 2, (June 1982), 159-192.
2.Bellman, K. L., Landauer, C. Designing testable, heterogeneous software environments. *J. Sys. Softw. 29*, 3, (June 1995), 199-217.
3.Benbasat, I., Dhaliwal, J. S. A framework for the validation of knowledge acquisition. Nowledge Acquisition 1*, 1, (1989), 215-233.*
4.Chang C., Combs, J. B., Stachowitz, R. A. A report on the expert systems validation associate (EVA). *Exp. Sys. With Appl. 1*, 3, (1990), 217-230.
5.Gupta, U.G. Validation and verification of knowledge-based systems: A survey. *J. Appl. Intell. 3*, (1993), 343-363.
6.Hayes-Roth, F., Jacobstein, N. The state of knowledge-based systems. *Commun. ACM 37*, 3 (March 1994), 26-39.
7.Howden, W.E. Introduction to Software Validation. *In Tutorial: Software Testing and Validation Techniques*. E. Miller, and W.E. Howden (Eds) IEEE Computer Society Press, New York, 1978, 1-2.
8.Kirani, S.H., Zualkernan, I.A., Tsai, W.T. Evaluation of expert system testing methods. *Commun. ACM 37*, 11 (Nov. 1994), 71-81.
9.Murphy, G.C., Townsend, P., Wong, P.S. Experiences with cluster and class testing. *Commun. ACM 37*, 9 (Sept. 1994), 39-47.

Uma G. Gupta (dcgupta@creighton.edu) is the Jack and Joan McGraw Endowed Chair in information technology management in the College of Business at Creighton University, Omaha, NB.

This research was conducted as part of the Intelligent Simulation Training System developed at the University of Central Florida and funded by the State of Florida.