N90-27335

AUTOMATED UNIT-LEVEL TESTING WITH HEURISTIC RULES

W. Homer Carlisle, Kai-Hsiung Chang,
James H. Cross, and William Keleher
Auburn University

Keith Shackelford
G. C. Marshall Space Flight Center

ABSTRACT

Software testing plays a significant role in the development of complex software systems. Current testing methods generally require significant effort to generate meaningful test cases. The QUEST/Ada* system is a prototype system designed using CLIPS (7) to experiment with expert system based test case generation. The prototype is designed to test for condition coverage, and attempts to generate test cases to cover all feasible branches contained in an Ada program. This paper reports on heuristics used by the system. These heuristics vary according to the amount of knowledge obtained by preprocessing and execution of the boolean conditions in the program.

INTRODUCTION

There are many approaches to software testing, and most require considerable human interaction at a great cost in man hours. The goal of automating this activity is to provide for more cost effective software testing and to avoid human bias or oversight. One class of automated testing tools, the dynamic analysis tools, is characterized by direct execution of the program under test (3). A test data generator is a dynamic analysis tool designed to assist the user in achieving goals such as statement coverage, condition coverage, or path testing. The difficulties of test data generation are due to the computation efforts, sometimes wasted, in computing infeasible paths or solving arbitrary path predicates, especially if a predicate contains non-linear terms or function calls. Consequently AI approaches must be utilized to avoid these problems.

QUEST/Ada* is a prototype system that is designed to experiment with expert system based test case generation. This system seeks to achieve its goals using heuristic rules to choose and generate new test cases. This paper reports on various rule sets designed to achieve condition coverage of Ada programs with increasing amounts of knowledge about the conditions in the Ada program. Knowledge can vary from little information about the input data (requiring random case generation of the appropriate type of input data), to complete symbolic solutions for variables in the conditions under test.

*Research and development of the QUEST/Ada system has been supported by the National Aeronautics and Space Administration (NASA). Ada is a trademark of the United States Government, Ada Joint Program Office.

BACKGROUND

Testing

The reliability of software is critical to space applications. One of the most common ways of ensuring software reliability is through program testing. There are three major categories of software testing: domain testing, functional testing and structural testing.

Domain testing

Programs run on finite state machines over finite input sets. Consequently it is theoretically possible to prove a program correct by testing it over its input domain. However in general these domains are too large for this type of testing to be feasible. It is therefore assumed that programs of arbitrary large storage requirements run on machines of arbitrary large size and precision. Unfortunately this assumption leads to results that demonstrate the impossibility of an algorithm to determine correctness of a program (4).

Functional testing

Functional testing is the process of attempting to find discrepancies between the program's output and its requirements specification (6). In functional testing (1) (4) a program is executed over selected input and the results are compared with expected output. Normally nothing is assumed about the internal structure of the program. Rather, test cases are constructed from knowledge of "what the program is supposed to do", i.e. its "function". This is known as the "black box" approach to testing

.

Structural testing

Structural or "white box" testing uses the source code control structure of a program to guide the selection of test data (1). One metric for the selection process is coverage, which is concerned with the number of structural units exercised by a test case. Examples of this metric are

Statement Coverage -	execute all statements in the program graph;
Branch Coverage -	encounter all exit branches for each decision node in the program graph;
Path Coverage -	traverse all paths of the graph.

Attempts to develop a practical test generation methodology for branch coverage have suggested approaches ranging from random test generation to full program path predicate solutions. Howden (4) has formalized test generation rules to help programmers test their code. Consequently such rules can be considered "expert knowledge" required for effective and automatic test case generation in an expert system test case generator.

Test case generation

The success of test data generation depends on knowledge of the internal structure of the program. Indeed, in the absence of any such knowledge, the only known testing method is random generation of test data and probabilistic determination of the equivalence of the function under test with desired behavior. On the other hand, if the structure of the program is well understood then by testing, complete validation over a limited domain may be possible. Consider for example a program consisting of a single input variable containing only assignment and increment operations. Such a restriction of a program determines that it can only compute a constant function $f(x) = c$ or a linear function $f(x) = x + c$ for some constant value c. With this knowledge two test cases are consequently sufficient to identify and validate the program.

Branch coverage is currently regarded as a minimal standard of achievement in structural testing (5). Thus, the goal of an expert system test case generator is to achieve branch coverage by using heuristic rules with execution feedback to generate test cases sufficient to insure that each branch in a program is invoked at least once. Figure 1 gives a system overview of such a test case generation methodology.

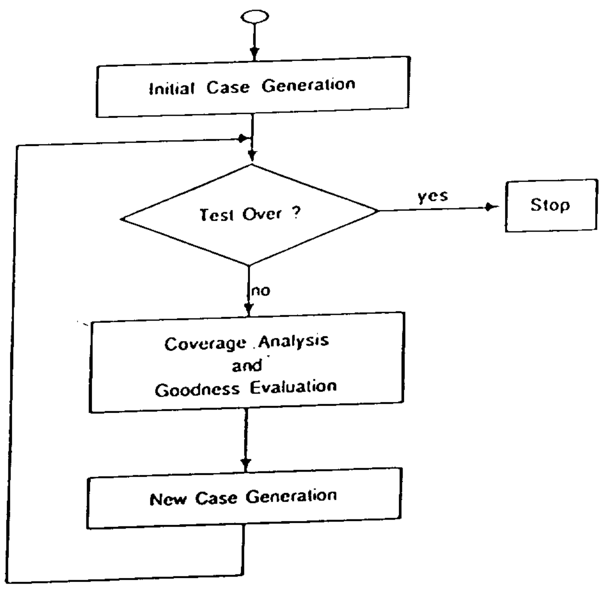

Figure 1

To avoid exponential searches, the analysis may be supported by a search strategy such as that proposed by Prather and Myers (5). This strategy views a software package as a flowgraph with each condition containing a true and false branch. The goal for test cases is to maximize the number of covered branches as recorded in a branch coverage table. The strategy is to select the first condition in a path from the start for which the condition has not yet been tested in both directions, and to generate (if possible) a test case that will drive this condition in the other direction. The idea behind this strategy is that, since some previous test case has reached the condition, it is already "close" to a test value required to drive an alternate branch of the condition.

AN INTELLIGENT TEST DATA GENERATION SYSTEM

QUEST/Ada is a prototype automated software testing tool presently implemented to support expert system based coverage analysis. The framework of QUEST/Ada will however support other rule based testing methods. Figure 2 gives an overview of the relationships among the major components of the system. An instrumented Ada module is supplied as input to a parser scanner that gathers information about the conditions being tested. Using compiled output of the parser/scanner, the test coverage analyzer executes the program for a test case and analyses the result. Based on this analysis, the test data generator uses rules to create new values for variables that are global to or are parameters to the unit under test. These variables are called "input variables".

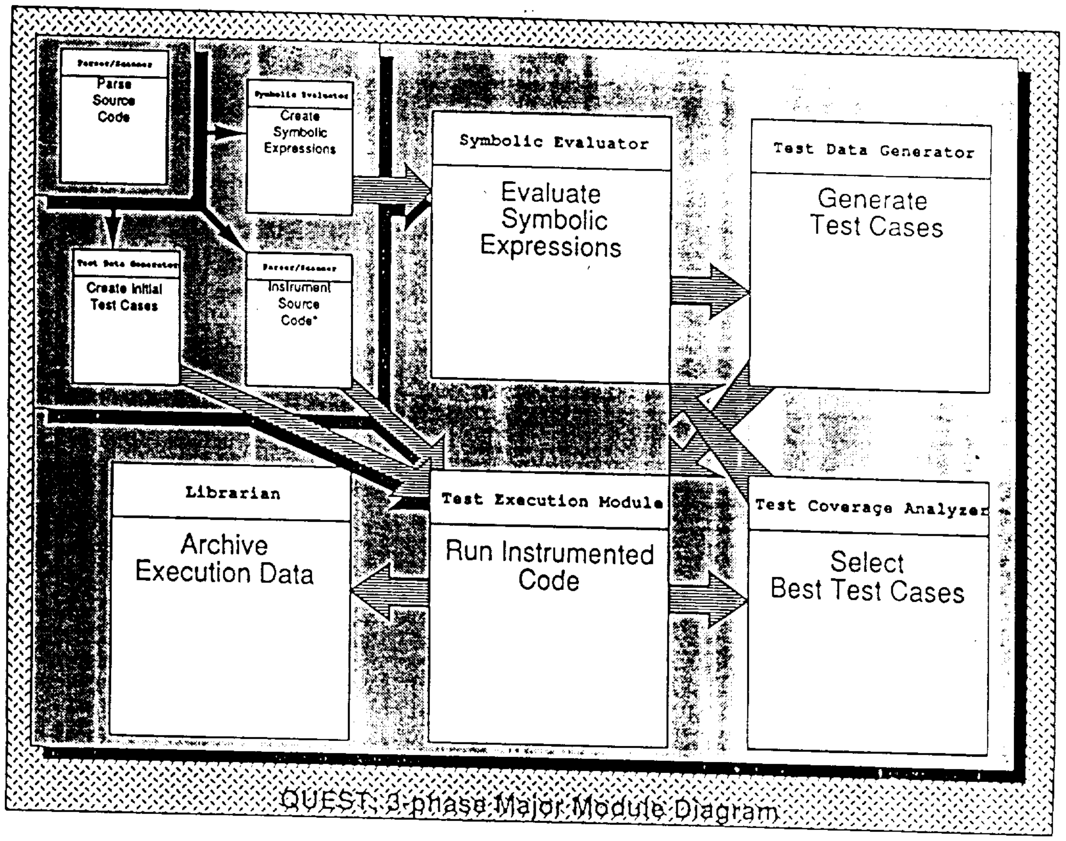

Figure 2

Initial test cases are needed to start the process. These may be provided by the user or generated by the system using an initial test case generation rule. Upon execution of the program on test cases, coverage analysis determines what branches have been covered and which branches need further testing. Coverage analysis is basically a table filling process recording the execution of each condition of the program. The expert system generates new test cases by applying rules based on knowledge about both the conditions not yet fully covered, and previous conditions in the execution path that lead to the condition not fully covered. New test cases are generated, and the testing continues. Execution stops when full coverage is indicated, or when a test case limit is reached. Implementation details of the QUEST/Ada system are described in (2).

Rule Based Test Case Generation

As designed, the QUEST/Ada system's performance is determined by the initial test case, rules chosen to generate new test cases, and the method used to select a best test case when there are several test cases that are known to drive a path to a specific condition.

Initial cases

If the user does not supply an initial test case, then initial test cases are generated by rules that require knowledge of the type and range of the input variables. For these variables test cases are generated to represent their mid-range, i.e. (upper-limit - lower-limit)/2, lower and upper values.

Best test case selection

When there are several test cases that drive a condition in a particular way, a rule is used to select from among these test cases a best test case. Experiments are being conducted with two "best test case" selection rules, with the second rule intended to be more knowledgeable than the first. In the first rule, the best test case represents a measure of the closeness of the left hand side (LHS) and the right hand side (RHS) of the condition as determined by the formula

$$|LHS - RHS|/(2*MAX(|LHS|,|RHS|).$$

The idea is that test values closer to the boundary of the condition are better. Problems arise in the search algorithm's attempt to cover all branches when a change in values of input variables change an execution path, and execution no longer reaches the condition. In order to decrease the likelihood of such unanticipated branching, a second approach to best test case selection has been designed. This approach utilizes information about the conditions in the execution path leading to the condition under consideration. In this situation, the formula for best test case selection takes into account the closeness of previous conditions. The heuristic idea is that for previous conditions in the execution path, the left hand side and right hand side of these conditions should be further apart. This heuristic assumption is based on the idea that small changes in the values affecting the condition under consideration will have a smaller impact on previous conditions when the left hand side and right hand side are far apart.

As an example, if two conditions c_1,c_2 precede condition c_3 in the execution path, and t_1, t_2,t_3 represent the "closeness" values associated with a test case t, then for weights w_1,w_2,w_3 a value determined by

$$w_3*t_3 + w_2*(1/t_2) + w_1*(1/t_1)$$

represents a better measure of the test case than does the value t_3. Note that the values of t_1,t_2,t_3 are in [0,1].

In general, if $c_1, c_2, \ldots c_{n-1}$ represent a path of conditions leading to a condition c_n, and for each i = 1..n

$$t_i=|LHS \text{ of } c_i - RHS \text{ of } c_i|/2*max(|LHS \text{ of } c_i|,|RHS \text{ of } c_i|)$$

then for some weights $w_1, \ldots w_n$, the best test case for condition n is chosen by a minimum value of

$$v = w_n*t_n + w_{n-1}/t_{n-1} + \ldots + w_1/t_1.$$

For testing in QUEST, weights of 1 for w_n and 1/(n-1) for $w_1 \ldots w_{n-1}$ were chosen.

Test case generation

In order to experiment with the effects of altering the knowledge about the conditions of a program under test, three categories of rules have been selected. The rules are in the syntax of "CLIPS" (7), a forward chaining expert system tool used by the QUEST/Ada prototype. Comments (lines beginning with ;) are intended to explain the action of the rule. The first category of rule reflects only "type" (integer, float, etc.) knowledge about the variables contained in the conditions. These rules generate new test cases by randomly generating values. The following listing provides an example of this type of rule.

Listing 1.

```
(defrule generate_random_test_cases ""
  (types $?type_list)
```

```
;use only type and
  (low_bounds $?low_bounds_list)
;boundary info
  (high_bounds $?high_bounds_list)
;to avoid run error
=>
;set up a loop to generate n test cases for the
;n input variables
  (bind ?outer_pointer 1)
  (while (<= ?outer_pointer (length $?type_list))
;get test case number
    (bind ?test_number (test_number))
    (format test-case-file " %d" ?test_number)
;step thru each variable
    (bind ?inner_pointer 1)
    (while (<= ?inner_pointer (length $?type_list))
;get the type of the variable
      (bind ?type       (nth ?inner_pointer $?type_list))
;assign it a random value
      (bind ?random_value (rand()))
;get range information
      (bind ?low_bound
                (nth ?inner_pointer $?low_bounds_list))
      (bind ?high_bound
                (nth ?inner_pointer $?high_bounds_list))
;be sure random value is within bounds
      (if (> ?random_value ?high_bound) then
        (bind ?test_value
           (* (/ ?high_bound ?random_value) ?high_bound))
       else
        (bind ?test_value ?random_value))
      (if (< ?random_value ?low_bound) then
        (bind ?test_value
           (* (/ ?low_bound ?random_value) ?low_bound))
       else
        (bind ?test_value ?random_value))
;write value for the variable to the test case file
;in appropriate format
      (if (eq ?type int) then
        (format test-case-file " %d" ?test_value))
      (if (eq ?type fixed) then
        (format test-case-file " %f" ?test_value))
      (if (eq ?type float) then
        (format test-case-file " %e" ?test_value))
;next variable in test case
```

```
        (bind ?inner_pointer (+ ?inner_pointer 1)))
      (fprintout test-case-file crlf)
;next test case
      (bind ?outer_pointer (+ ?outer_pointer 1)))
)
```

The second category of rule attempts to incorporate information that is routinely obtained by a parse of the expression that makes up a condition (such as "type" and "range"), information about coverage so far obtained, and best test cases for previous tests. This particular example uses the best test case associated with a condition, and for n input variables, generates n test cases by altering each variable one percent of its range. Listing #2 gives and example of this category of rule.

Listing 2.

```
(defrule generate_increment_by_one_percent_test_cases ""
  (types $?type_list)
  (low_bounds $?low_bounds_list)
  (high_bounds $?high_bounds_list)
;match any condition that is only half covered
  (coverage_table ?decision ?condition true|false)
;get the best test case for each condition
  (best_test_case ?decision ?condition $?values)
=>
  (bind ?outer_pointer 1)
  (while (<= ?outer_pointer (length $?values))
    (bind ?test_number (test_number))
    (format test-case-file " %d" ?test_number)
    (bind ?inner_pointer 1)
    (while (<= ?inner_pointer (length $?values))
      (bind ?type       (nth ?inner_pointer $?type_list))
      (bind ?high_bound
              (nth ?inner_pointer $?high_bounds_list))
      (bind ?low_bound
              (nth ?inner_pointer $?low_bounds_list))
;increment the current variable by one percent of
;its range
      (bind ?one_percent (/ (- ?high_bound ?low_bound) 100))
      (bind ?increment
            (+ (nth ?inner_pointer $?values) ?one_percent))
;if this is the variable we want to alter
      (if (= ?outer_pointer ?inner_pointer) then
        (if (<= ?increment ?high_bound) then
          (bind ?test_value ?increment)
         else
          (bind ?test_value ?low_bound))
       else
;and the other variables are written as is
        (bind ?test_value (nth ?inner_pointer $?values)))
      (if (eq ?type int) then
        (format test-case-file " %d" ?test_value))
      (if (eq ?type fixed) then
        (format test-case-file " %f" ?test_value))
      (if (eq ?type float) then
        (format test-case-file " %e" ?test_value))
      (bind ?inner_pointer (+ ?inner_pointer 1)))
    (fprintout test-case-file crlf)
    (bind ?outer_pointer (+ ?outer_pointer 1)))
)
```

The final type of rule utilizes information about the condition that can be obtained by symbolic manipulation of the expression. The given rule uses a boundary point for input variables associated with the true and false value of a condition. This value is determined by using symbolic manipulation of the condition under test. Many values can be chosen that cross the boundary of the condition and, as with best test case selection, we seek to choose a value that will not alter the execution path to the condition. In addition to best test case selection we now have additional knowledge to generate new test cases. We use the values of variables at a condition and compare them with values of the variables that reach the condition. This added information is incorporated in the generation of new test cases. To achieve this, the following approach has been taken by the above rule.

Suppose that for an input variable x appearing in a condition under test, the value of x at the condition boundary has been determined to be x_b and the input value that has driven one direction of the condition has been x_i. Although we do not know how x is modified along the path leading to the condition (the value of x on input may be expected to differ from the value of x at the condition) we are able to establish that the value of x at the condition is x_c. In this situation we choose as new test cases (provided the values lie in the limits allowed for values of x)

$$x_b*(x_i/x_c) + e$$

where e is 0 or takes on a small positive or negative value. Listing 3 is an example of this heuristic.

Listing 3.

```
(defrule generate_symbolic_approximation_plus_increment_test_cases ""

;type information here
  (types $?type_list)
  (low_bounds $?low_bounds_list)
  (high_bounds $?high_bounds_list)
;knowledge about the condition here
  (coverage_table ?decision ?condition true|false)
  (best_test_case ?decision ?condition $?values)
  (value_at_cond ?decision ?condition $?vacs)
  (symbolic_boundary ?decision ?condition $?boundaries)
```

```
=>
  (bind ?outer_pointer 1)
  (while (<= ?outer_pointer (length $?values))
    (bind ?test_number (test_number))
    (format test-case-file " %d" ?test_number)
    (bind ?inner_pointer 1)
    (while (<= ?inner_pointer (length $?values))
      (bind ?type   (nth ?inner_pointer $?type_list))
;for the variable under consideration
      (if (= ?outer_pointer ?inner_pointer) then
;for its range
        (bind ?high_bound
               (nth ?inner_pointer $?high_bounds_list))
        (bind ?low_bound
               (nth ?inner_pointer $?low_bounds_list))
;get its input value
        (bind ?   (nth ?inner_pointer $?values))
;and its value at condition
        (bind ?Xc  (nth ?inner_pointer $?vacs))
;and the boundary of the condition
        (bind ?Xb  (nth ?inner_pointer $?boundaries))
;generate a guess as to an input value leading to boundary
        (bind ?approximation (* (/ ?Xi ?Xc) Xb))
;generate a small amount to move around boundary
        (if (< (abs ?high_bound) (abs ?low_bound)) then
          (bind ?small_bound ?high_bound)
         else
          (bind ?small_bound ?low_bound))
        (bind ?digit 0)
        (while (!= (trunc ?low_bound) ?low_bound)
          (bind ?digit (+ ?digit 1))
          (bind ?low_bound (* ?low_bound (** 10 ?digit))))
;call it e
        (bind ?e (** 10 (* -1 ?digit)))
        (bind ?incremented_approximation
;increment the approximation by e
                    (+ ?approximation ?e))
        (if (<= ?incremented_approximation ?high_bound) then
          (bind ?test_value ?incremented_approximation)
         else
          (bind ?test_value ?high_bound))
       else
        (bind ?test_value (nth ?inner_pointer $?values)))
;write to test case file in appropriate format
      (if (eq ?type int) then
        (format test-case-file " %d" ?test_value))
      (if (eq ?type fixed) then
        (format test-case-file " %f" ?test_value))
      (if (eq ?type float) then
        (format test-case-file " %e" ?test_value))
      (bind ?inner_pointer (+ ?inner_pointer 1)))
    (fprintout test-case-file crlf)
;next test case
    (bind ?outer_pointer (+ ?outer_pointer 1)))
)
```

CONCLUSION

The objective of the research has been to achieve more effective test data generation by combining software coverage analysis techniques and artificial intelligence knowledge based approaches. The research has concentrated on condition coverage and uses a prototype system built for expert system based coverage analysis. The success of this approach depends on the search algorithm used to achieve coverage and the heuristic rules employed by the search. The effectiveness of rules vary according to the knowledge about the source and the knowledge obtained by previous test cases. The QUEST/Ada prototype provides an extendible framework which supports experimentation with rule based approaches to test data generation. In particular it facilitates the comparison of these rule based approaches to more traditional techniques for ensuring software test adequacy criteria such as branch coverage, and allows for modification and experiments with heuristics to achieve this goal.

REFERENCES

(1) Beizer, B., *Software System Testing and Quality Assurance*, New York: Van Nostrand Reinhold Co., 1984.

(2) Brown, David B., *Quest/Ada: Query Utility Environment for Software Testing of Ada*, Phase 1 Report, Contract NASA-NCC8-14, Department of Computer Science and Engineering, Auburn University, Alabama, 1989.

(3) DeMillo, R.A., McCracken, W.M., Martin, R.J., Passafiume, J.F., *Software Testing and Evaluation*, Benjamin/Cummings Publishing Co., Inc. Menlo Park, California, 1987.

(4) Howden, W.E., *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

(5) R.E. and Myers, P.,Jr., *The Path Prefix Software Testing Strategy*, IEEE Transactions on Software Engineering, Volume SE-13, Number 7, July 1987, pp. 761-765.

(6) Meyers, G.J., *The Art of Software Testing*, John Wiley and Sons, New York 1979.

(7) National Aeronautics and Space Administration, *CLIPS Reference Manual*, Artificial Intelligence Section Johnson Space Center, Version 4.1 September 1987.