QUEST/Ada

QUERY UTILITY ENVIRONMENT FOR SOFTWARE TESTING OF ADA

The Development of a
Program Analysis Environment
for Ada

Contract Number NASA-NCC8-14

Task 1, Phase 3 Report
(First Six Months)

Department of Computer Science and Engineering
Auburn University, Alabama 36849-5347

Contact: David B. Brown, Ph.D., P.E.
Professor and
(205) 844-6314
brown@auburn.eng.edu

February 1991

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

We appreciate the assistance provided by NASA personnel, especially Mr. Keith Shackelford whose guidance has been of great value. Portions of this report were written by each of the members of the project team. The following is an alphabetized listing of project team members.

FACULTY INVESTIGATORS

Dr. David B. Brown, Principal Investigator
Dr. W. Homer Carlisle
Dr. Kai-Hsiung Chang
Dr. James H. Cross

GRADUATE RESEARCH ASSISTANTS

Eric Richards
Benjamin B. Starke
Kevin Sullivan

UNDERGRADUATE RESEARCH ASSISTANT

Ken Kirksey

1.0 EXECUTIVE SUMMARY

This report presents the results of research and development efforts of the first six months of Task 1, Phase 3 of a general project entitled "The Development of a Program Analysis Environment for Ada." The scope of this task was defined early in Phase 1 (initiated June 1, 1988) to include the design and development of a prototype system for testing Ada software modules at the unit level. The system was called Query Utility Environment for Software Testing of Ada (QUEST/Ada). The report for Task 2 of this project, entitled "Reverse Engineering Tools for Ada Software," is given in a separate volume, since the documentation activities for Task 1 and Task 2 are being conducted independently.

Phase 1 of this task completed the overall QUEST/Ada design, which was subdivided into three major components, namely: (1) the parser/scanner, (2) the test data generator, and (3) the test coverage analyzer. A formal grammar specification of Ada and a parser generator were used to build an Ada source code instrumenter. Rule-based techniques provided by the CLIPS expert system tool were used as a basis for the expert system. The prototype developed performs test data generation on the instrumented Ada program using a feedback loop between a test coverage analysis module and an expert system module. The expert system module generates new test cases based on information provided by the analysis module. Information on the design is given in the Phase 1 Report [Brown89], and these details will not be repeated here.

The goals of Phase 2 were: (1) to continue to develop and improve the current user interface to support continued research, (2) to develop and empirically evaluate a succession of alternative rule bases for the test case generator such that the expert system achieves coverage in a more efficient manner, and (3) to extend the concepts of the current test environment to address the issues of Ada concurrency. In Phase 2 the entire system was ported to the UNIX operating system operating on a Sun SPARCstation network under XWindows. This, along with the addition of a Librarian module within QUEST, greatly improved its usability; however, additional work on the user interface was still required at this point. The evolution of QUEST concentrated upon the Test Data Generator (TDG) module, which is the expert system designed to select the test data that will be most likely to drive a specific control path in the program. Four types of rules were used in the development of the TDG: random, initial, parse-level, and symbolic evaluation. Finally, a major literature review was conducted on the subject of testing Ada concurrency constructs, and an approach was formulated for integrating this into QUEST. Details of activities within Phase 2 are presented in the Phase 2 report [Brown90].

The goals of Phase 3 are: (1) to further refine the rule base and complete the comparative rule base evaluation, (2) to implement and evaluate a concurrency testing prototype, (3) to convert the complete (unit-level and concurrency) testing prototype to a workstation environment, and (4) to provide a prototype development document to facilitate

the transfer of the research technology to a working environment. These goals have been partially met in the first six months of Phase 3 as summarized in the following paragraphs.

To a large extent developments within this period with regard to refining the prototype have been driven by the example Ada code modules which were obtained from NASA. Since the previous version of the prototype had only considered integer types, fixed and floating point types were added to the test data generation rules. This innovation is described in Section 2.1. Also, provisions were made to handle multiple conditions and global variables, which are described in Sections 2.3 and 2.4.

Work in the innovation of the test case generator took the form of implementing the design work that was done in Phase 2. A condition-oriented two-phased approach was taken, which is condition-oriented in the sense that only partially-covered conditions are considered for further new case generation. During the first phase of the process, a best test case is selected as the model for new cases. Two measurements were designed to select the best test case. One measurement is based solely on the target condition while the other measurement is based on the target condition plus the conditions that are on the path to the target condition. During the second phase, one of the three developed methods is selected to generate new cases based on the given best case. The three methods of modification are random, fixed percentage, and symbolic evaluation. A detailed description of these innovations are described in Section 2.2.

Actual execution of the updated prototype enabled the comparison of the test case generation rules with those previously applied, both for the example modules which were previously used to test QUEST, and some of the NASA-supplied modules. The results of these experiments, given in Section 3, were mixed. Some showed the newly implemented approach to be quite good, while others showed them to be about the same as previous methods. Since these were performed late in the reporting period, a detailed analysis of these results has not been performed. These results will be invaluable in guiding the remaining part of the project.

The porting of the prototype to a workstation environment (in particular, Sun SPARCstation, UNIX, XWindows) has been completed, and a technology transfer document has been drafted. Additional work needs to be done, however, in enabling the prototype to take full advantage of this environment. In particular, the ability to execute the module under test without leaving the QUEST environment is a feature which will be added along with other improvements in the user interface during the remainder of Phase 3. Also, the technology transfer document given in Section 3 should be regarded as a very preliminary draft at this point.

The effort with regard to concurrency will be given additional emphasis in the remainder of the project. At this point additional design effort has been directed at evaluating the alternatives proposed at the end of Phase 2. It has been determined that the lock-step/monitor approach has the greatest chance for success, and a discussion of this is

given in Section 4. Due to the emphasis upon the other prototype modifications, however, no accommodations for this have yet been made in the prototype.

2.0 PROTOTYPE DEVELOPMENT

2.1 FIXED AND FLOATING POINT EXTENSIONS

The test data generation rules have been extended to handle fixed point and floating point variable types. Information on all variables is passed from the code analyzer to the test data generator through the file FACTOR.CLP. This file contains a "define facts" statement for use by CLIPS. For example,

```
(deffacts
    (names x y z)
    (types int fixed float)
    (deltas 0 0.01 0)
    (low-bounds -5000 -50000 -100000)
    (high-bounds 5500 50000 100000))
```

indicates that in the module under test, there are three variables, x, y, and z, and their data types, deltas, low bounds, and high bounds are as shown. The delta value for a fixed point variable is the distance between possible values for that variable. Delta values for variables that are not fixed point types have no meaning in this context.

CLIPS supports only one numeric type -- number. Consequently there are certain operations that are not supported, e.g., integer division. CLIPS stores all numbers as single precision floating point values, which may be a source of round off and overflow errors. This limitation could be eliminated by rewriting and recompiling some of the CLIPS source code, but it was decided that this would not be necessary for the prototype implementation.

Another difficulty encountered with CLIPS is its inability to read numbers which are not in the following format: an optional sign (+ or -), digits (0-9), an optional decimal point and digits, and an optional e for exponential notation with a corresponding (optional) sign and digits for the exponent. For example 237, 15.09, +12.9, 3e5, and -32.3e-7 are all valid numbers. If the module under test contains constants using different notations (such as floating point numbers containing X's), these must be modified to be compatible with CLIPS. In order to compensate for this, the formats of the numbers output by the test data generator are determined by the types of the corresponding variables, as determined from the FACTOR.CLP file. Integer test data are output in integer format, fixed point data are output in CLIPS' floating point format, and floating point data are output in floating point format using power of ten notation.

2.2 INNOVATIONS IN TEST CASE GENERATION

The objective of typical test case generation is to generate test cases that provide maximal software coverage. Coverage here is measured in terms of the completeness of driving condition branches. In order to achieve this goal, a condition-oriented two-phase approach was developed [Brown90]. It is condition-oriented in the sense that only partially-covered conditions (called target conditions) are considered for further new case generation. During the first phase of the process, a *best test case* is selected as the model for new cases. Two measurements have been designed to select the best test case. One measurement is solely based on the target condition. The other measurement is based not only on the target condition but also on the conditions that are on the path to the target condition. During the second phase, one of the three developed methods is selected to generate new cases based on the given best case. The three methods are random modification, fixed percentage modification, and symbolic evaluation based modification.

Although the design of the test case generation approach was reported in an earlier report [Brown90], two parts of the design were not fully implemented. One of these was the best test case measurement considering conditions on the path, while the second was the fixed percentage modification. In the earlier prototype, the amount of modification was based on the *value* of a variable. The updated version of the prototype uses the declared range of a variable as the basis. A complete and updated design is presented here to provide a self-explanatory document.

2.2.1 TEST CASE GENERATION STRATEGY

The objective of this framework is to achieve maximal branch coverage. In order to ensure fruitful test case generation, a branch coverage analysis is needed. The coverage analysis follows the Path Prefix Strategy of Prather and Myers [PRA87]. In this strategy, the target source code is represented as a simplified flow chart. The branch coverage status of the code is recorded in a coverage table. When a branch is driven (or covered) by any test case, the corresponding entry in the table is marked with an "X". The goal of the test case generation is to mark all entries in the table.

Consider Figures 2.2.1a and 2.2.1b. Currently, conditions 1 and 2 are fully covered; conditions 3, 4, and 5 are partially covered; and condition 6 is not covered. Since conditions 1 and 2 are fully covered, there is no need to generate more cases for them. Condition 3, on the other hand, is partially covered. More cases should be generated to drive its false branch, i.e., 3F, which is not yet covered. The Path Prefix Strategy states that new cases can be generated by modifying a test case, say case-3T, that drives branch 3T. Consider the fact that case-3T starts at the entry point and reaches condition 3. Although it drives 3T, it is "close" to driving 3F. Slight modification of case-3T may devise some new cases that will drive 3F.

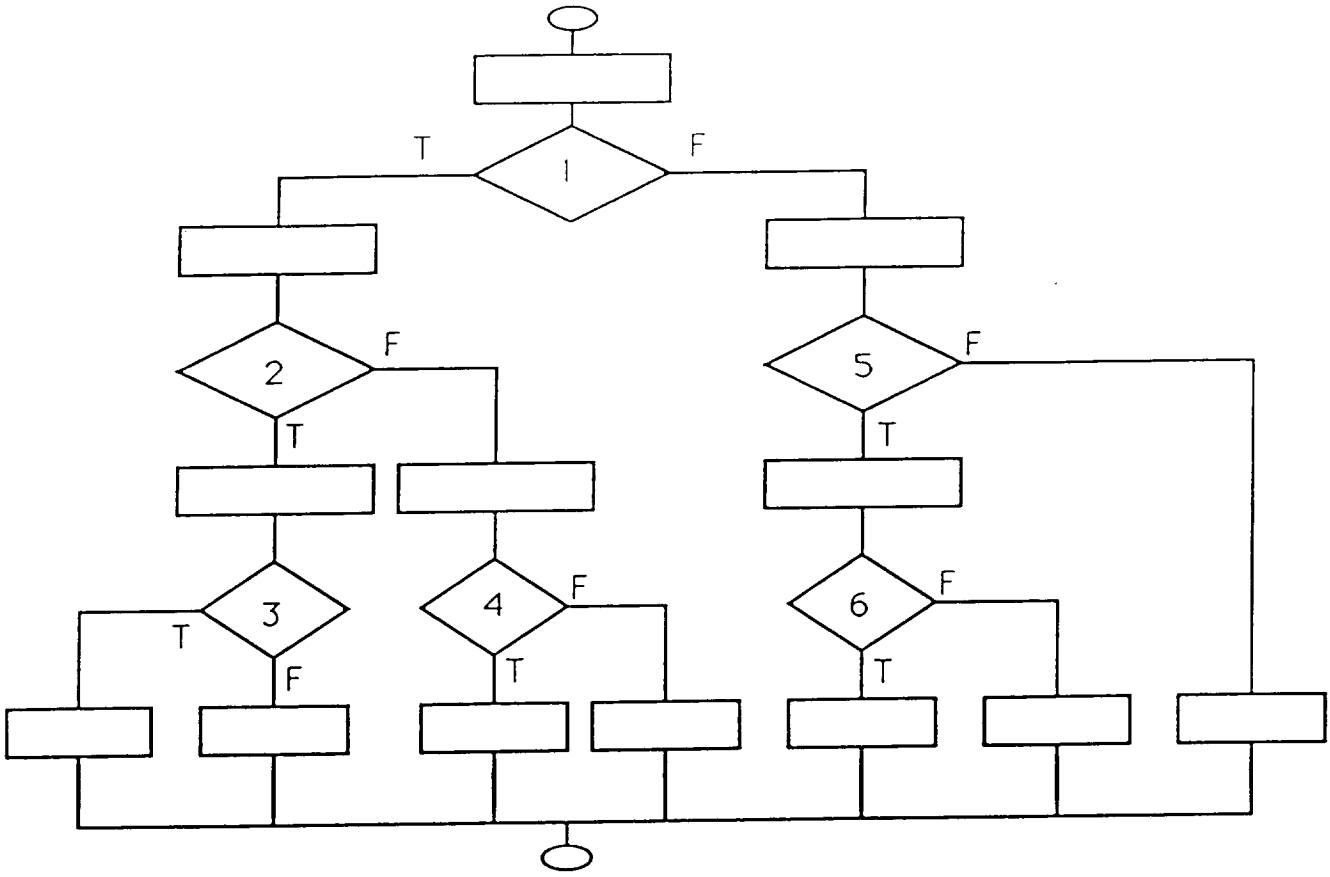

Figure 2.2.1a A sample flow chart

Condition	Branch T	Branch F
1	X	X
2	X	X
3	X	
4	X	
5		X
6		

Figure 2.2.1b Coverage table of Figure 2.2.1a

With this strategy in mind, the test case generator should target partially covered conditions. Earlier test cases can be used as models for new cases. Conditions that have not been reached yet, e.g., condition 6 in Figure 2.2.1b, will not be targeted for new case generation. This is because no test case model yet exists that can be used for modification. A test case model will eventually surface later in the process, and in this example, after condition 3, 4 and 5 are fully covered, a model for condition 6 will appear.

2.2.1.1 BEST TEST CASE

Problems arise when there is more than one test case driving the same path. For example, if cases 1, 2, ..., n all drive branch 3T of Figure 2.2.1b, then the selection of a model case for branch 3F becomes problematic. It is necessary to quantify the "goodness" of each case and use the "best" case as the model for modification.

Consider the typical format of an IF-THEN statement: IF exp THEN do-1 ELSE do-2. The evaluated Boolean value of exp determines the branching. Exp can be expressed in the form of: LHS <op> RHS. The goodness of a test case, t1, relative to a given condition can be defined as

$$| \text{LHS}(t1) - \text{RHS}(t1) | \; / \; (2 * \text{MAX} (| \text{LHS}(t1) |, | \text{RHS}(t1) |)) \qquad (1)$$

LHS(t1) and RHS (t1) represent the evaluated value of LHS and RHS, respectively, when t1 is used as the input data. This measure tells the closeness between LHS and RHS [DEA91]. When this measure is small, it is generally true that a slight modification of t1 may change the truth value of exp, thus covering the other branch. The importance of slight modification to a model test case is based on the fact that the model case starts from the entry point and reaches the condition under consideration. Between the entry point and the condition, the modified cases must pass through exactly the same branching conditions and yield the same results. For this reason, the smaller the modification is, the better the chance will be for a modified case to stay on the same path. The measurement of (1) provides this "goodness" of a test case which ranges from 0 to 1. A test case that yields the smallest measurement is considered to be the best test case of the condition under consideration.

The closeness measurement has a serious risk. Recall that a set of new test cases is generated based on the best test case of a partially covered condition (called target condition), and the intent of the new test cases is to cover the uncovered branch of the target condition. This closeness is computed based on the target condition only. A slight modification to the target condition may not have the same meaning to those conditions on the path. This may result in what we will call unanticipated branchings along the path i.e., a flow of control that may no longer drive the target condition. In order to reduce the likelihood of unanticipated branching, a test case's goodness measure should also consider those conditions that are on the path leading to the target condition. This idea can be expressed in the following example.

In Figure 2.2.1c, two test cases, t_a and t_b, pass through the false branches of conditions 1, 2, and 3, of Figure 2.2.1a. Assume the goal is to generate more cases to cover the truth branch of condition 3. Either t_a or t_b should be used as the model for the new cases. If the whole input space is represented as R, it can be divided into several subspaces (see Figure 2.2.1c). First, R is divided into 1T and 1F, which represent the portions of input space that drive the truth and false branches of condition 1, respectively. Similarly, 1F can be divided into 2T and 2F, and 2F can be divided into 3T and 3F.

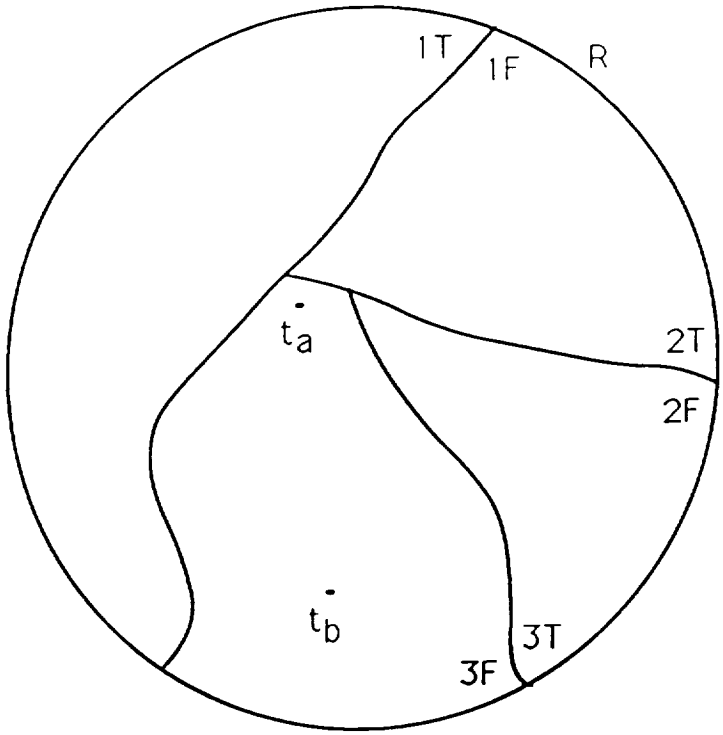

Figure 2.2.1c Inpurt space of the program in figure 2.2.1a

In this example, both t_a and t_b fall within the subspace of 3F. A best test case must be selected between t_a and t_b for new case generation. According to the earlier definition, the goodness is related to the distance that each test case is from the boundary of 3T and 3F. Based on this definition, t_a is closer to the boundary so it should be chosen as the best test case. From the viewpoint of condition 3, this is correct. A relatively small modification to t_a may lead to 3T. However, t_a is also close to the boundaries of conditions 1 and 2. There is a good chance that a slight modification to t_a may lead to undesired branching at conditions 1 and 2.

We will call the modification magnitude that is required to drive a different branch at a condition the freedom space of a test case. In this example, t_a has a small freedom space at condition 3 which is desirable. But its freedom spaces at conditions 1 and 2 are also small, which may cause unanticipated branchings. On the other hand, although t_b is not as close to condition 3's boundary as t_a is, it is not close to any other boundaries either. A larger modification may be required for t_b to lead to 3T. Since t_b is far away from any other boundaries, a larger modification may not cause any unanticipated branches. For this reason, the goodness of a test case concerning a target condition should be determined by the freedom space at the target condition as well as the freedom spaces of all conditions that are on the path to the target condition. For the former element, the smaller the better; for the latter element, the larger the better. The goodness can now be redefined as:

$$G(t,D) = w * L(t,D) + (1-w) * P(t,D) \qquad (2)$$

where:

G(t,D) : Goodness of test case t at condition D.
L(t,D) : Freedom space of t at D.
P(t,D) : Sum of freedom space reciprocals of t along the path toward D.
w : Weighting factor between L(t,D) and P(t,D), $0 < w < 1$.

L(t,D) is defined as in formula (1), and P(t,D) is defined as:

$$P(t,D) = \sum_{\text{all } D_i} 1 / (n * L(t,D_i)) \qquad (3)$$

Here, D_i is a condition that is on the path toward D, and n is the total number of these conditions. Although this definition does not represent the actual distance of test case t to a boundary, it is a reasonable approximation. With this definition, the smallest value indicates the best test case. Although formula (2) seems more appropriate than formula (1), it is difficult to prove this theoretically since both methods are heuristic. Both definitions are derived heuristically.

2.2.1.2 TEST CASE GENERATION PROCEDURE

The basic idea of new case generation is to modify the best test case of a target condition slightly with the intent to drive the uncovered branch of the condition. In Figure 2.2.1.2, input to the procedure contains three parameters x, y, and z. Assume condition D's truth branch is covered, and its best test case is (x_1, y_1, z_1). More cases must be generated to cover D's false branch. Condition D can be expressed as LHS(x, y, z, v_1, v_2, ...) <op> RHS(x, y, z, v_1, v_2, ...). Here, v_1,v_2,... are internal variables of the procedure. Input parameters x, y, and z may or may not be modified between the entry point and condition

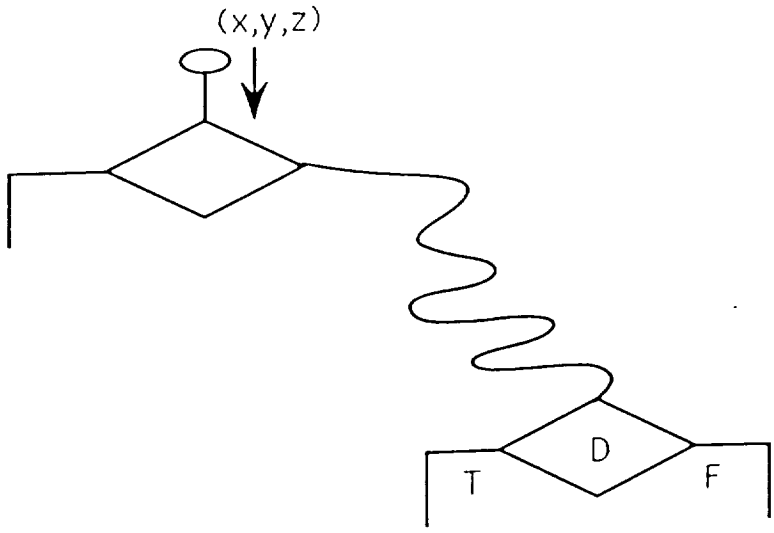

Figure 2.2.1.2 A test case (x,y,z) drives condition D

2.2.1.2.1 INCREMENT AND DECREMENT MODIFICATION

This method increments and decrements each parameter of the best test case with a fixed percentage of each parameter's ranges. The percentage can be any one of or any combination of 1%, 10%, 20%, 40%, etc. For example, if the best test case is (x_1, y_1, z_1) and the ranges for input variables x, y, and z are [0 10], [-100 0], and [-50 50] respectively, a 1% increment and decrement would generate new cases like (x_1+0.1, y_1+1, z_1+1) and (x_1-0.1, y_1-1, z_1-1).

2.2.1.2.2 BOUNDARY COMPUTATION

This approach finds the boundary that separates the truth and the false values of a condition, say D. It then tries to modify the best case to cover both sides of the boundary. Since the branching of D can only be externally controlled by input parameters, the condition boundary should be defined for x, y, and z. For example,

x_b = f1 (y, z, v_1, v_2, ...)
y_b = f2 (x, z, v_1, v_2, ...)
z_b = f3 (x, y, v_1, v_2, ...)

These boundary equations can be derived from D using symbolic manipulation. For example, given a condition

x + 3 * y =< 4 - 6 * z + v

$$x + 3 * y \quad =< \quad 4 - 6 * z + v$$

the condition boundary will be

$$x_b = 4 - 6 * z + v - 3 * y$$
$$y_b = (4 - 6 * z + v - x) / 3$$
$$z_b = (4 - x - 3 * y + v) / 6$$

Remember that the new case generation should be based on the best case, (x_1, y_1, z_1), and the modification should be as small as possible. A simple strategy would be to modify only one variable at a time. For example, we can modify x and keep y and z unchanged. In order to compute the boundary value of x at D, the actual values of y, z, v_1, v_2, ... just before D should be used in the computation. The computation provides the desired boundary value of x at condition D, say x_b. Three new cases can be generated to cover both truth and false branches: (x_b, y_1, z_1), (x_b+e, y_1, z_1), and (x_b-e, y_1, z_1). Here, e is a small positive number, e.g., e = (range of x) / 100.

Up to this point, it is assumed that x (or y or z) would not be modified between the entry point and condition D. This may not be valid at all. If an input parameter is modified by the program before reaching the target condition, the precise computation of the boundary may lose its purpose. The question becomes: what can be done if an input parameter has been modified? If the desired boundary value of x at condition D is x_b, this value must be inverted back through the path that leads to condition D. Through this inversion, the value of x at the entry point can be found. However, this is a complex path predicate problem which does not have a general solution [PRA87].

Consider the following situation. The input value of x is x_i, the actual value of x just before condition D is x_c. Assume x has been modified before reaching D and the boundary value of x at D is x_b. We might surmise that input x should be changed from x_i to an unknown value x_u such that, just before reaching D, x will be changed from x_c to x_b. Since we do not know how x is modified along the path, precise modification to x at the entry point cannot be computed. However, an approximation can be derived. At condition D, the desired value of x is x_b and the provided value is x_c. We may consider x_i is off the target, i.e., the condition boundary at D, by the following percentage:

$$|x_b - x_c| / (2*MAX(|x_b|, |x_c|)) * 100\ \% \qquad (4)$$

Following this measurement, we can modify input x based on this percentage.

2.3 MULTIPLE CONDITIONS

A branching decision may contain boolean expressions of two or more conditions. In Ada, the binary boolean operators are AND, AND THEN, OR, and OR ELSE. The AND THEN and OR ELSE operators cause "short circuit" evaluations in that they do not necessarily evaluate all terms of the expression if a subset of them can determine the truth value of the expression. Currently the expert system component of the prototype test data generator handles only relational expressions as conditions (i.e. arithmetic expressions involving relational operations). In order to handle boolean operators within the prototype system, instrumentation transforms boolean operators into nested if statements. It is recognized that there are semantic differences between nested branching statements when the boolean operators have side effects. However, for simplification of the prototype system these issues are ignored. A condition number and decision number are passed as parameters to the instrumented procedure that replaces a decision to differentiate between the AND, AND THEN and OR, OR ELSE. For example, if the A and B are expressions involving a single relational operator, and the Ada code is:

```
IF A AND B THEN .... END IF;
```

the instrumented code will appear as:

```
condition_number := 1;
IF decision( decision_number, condition_number, relop(...A...) ) THEN ...
        condition_number := condition_number + 1;
        IF decision(decision_number, condition_number, relop(...B...) ) THEN ...
```

However, the Ada statement

```
IF A AND THEN B THEN .... END IF;
```

will be treated by the expert system as two decisions in order to seek coverage, as follows:

```
condition_number := 1;
IF decision(decision_number, condition_number, relop(...A...) ) THEN ...
        decision_number := decision_number + 1;
        IF decision(decision_number, condition_number, relop(...B...) ) THEN  ...
```

In order to more appropriately handle the semantics of the AND and OR operators, and to introduce rules into the rule base that have knowledge of these operators and which take advantage of this information, a special function called "boolop" will be developed which is analogous to the "relop" function employed for relative operators. Continuing to use the AND and AND THEN as examples, the instrumentation for the AND THEN operator is as previously described, that is, it is treated as two separate decisions. An if statement containing the AND of two relational expressions A and B will be instrumented with procedure call to a procedure to handle this operator, as shown below:.

IF boolop("AND",decision(..), decision(...)) THEN ...

In attempting to achieve coverage for this condition, the expert system rule set can take advantage of information about the sets of variables that occur in the expression A, the expression B, and those variables that occur in both expressions A and B. This information can be used in determining which variables should be altered in attempting to alter the boolean value of the condition of the if statement. For example, in attempting to drive the condition TRUE when the decision A is FALSE and B is TRUE, the next test case can avoid altering the values of input variables that have appeared in B. Or, if both A and B have a FALSE value, the TCG can seek to alter input variables appearing in both A and B in order to achieve the change in boolean value.

2.4 GLOBAL VARIABLES AND MODULE DEFINITIONS

Under the QUEST system design which evolved out of Phase 2 of this project, modules were generally considered to be independent with communication performed through the Ada parameter passing mechanism. Global variables were originally ignored in order to bring the prototype into operation as quickly as possible. In the process of attempting to test NASA-supplied code it was found that global variables were quite widespread. This section discusses modifications in the prototype which were required to accommodate global variables and the revised concept of a "module definition" which this necessitated.

2.4.1 GLOBAL VARIABLES

Variables that are global in scope must be treated as parameters to the test module. They can affect the functions of a module in the same way a formal parameter can, and similarly, they can be affected by the module. Several different methods for handling global variables were considered. The simplest approach is to treat all of the system's global variables as potential parameters to the system. This requires minimal design and implementation time, but the resulting test execution module would have to search through many more combinations of inputs than is necessary. If there were more than a few global variables in the system, this increases the test time more than is acceptable.

The ideal approach is to identify just those global variables that are referenced by the test module and by all of the procedures and functions that the test module calls. This particular subset of global variables would then be treated as parameters to the module. However, this method was determined to be infeasible because it would require too much time to design and implement.

The method chosen to update the prototype is a balance between the two approaches discussed above. The parser/scanner searches through the test module for all occurrences

of variables, and each global variable that appears within the module itself will be treated as a parameter. This will serve for the current prototype in that it is estimated to apply to approximately 75% of the modules to be tested. However, in any final implementation of QUEST, the implementors should consider the ideal case. This extension of the prototype will more accurately inform the Test Data Generator of the test module's inputs, which will result in a shorter testing time.

In order to implement this approach, the parser/scanner uses the DIANA package to find the global variables and their type definitions. DIANA is an intermediate representation for Ada programs and an interface to the Ada library. It retrieves compilation units from the Ada library and stores them as a net. Many of the activities described below are facilitated through DIANA.

The parser/scanner (P/S) retrieves the compilation unit for the requested test module as well as all of the compilation units which the test module depends upon. Then it searches through the module for each occurrence of any variable. If a variable does not appear in the module's local symbol table, it is assumed to be a global variable. Every compilation unit is then searched for the type of the global variable. The type information is then used to create the P/S facts which are sent to the Test Data Generator (TDG).

2.4.2 PACKAGES

The previous version of the prototype assumed that the module under test was not inside of a package. The improved prototype includes the ability to test modules inside of a package. This required the following two modifications:

1) The user/user interface was changed to specify the package name as well as the procedure or function name.

2) The parser/scanner was modified to recreate and compile the entire package when instrumenting.

2.4.3 DEFINITIONS OF SYSTEM, MODULE, AND TEST

QUEST must have access to all of the compilation units which make up each software system under test. QUEST will represent each system with a single directory which contains an Ada library. This library and it's associated Ada path will give access to all of the compilation units of the system.

The Module Under Test (MUT) must be instrumented, recompiled, and linked with a main procedure. The main procedure is a harness which reads test data and sets the data

up as parameters. The combined MUT and the harness make up the Test Execution Module. This module is placed in an Ada library in a sub-directory of it's system directory.

When the Test Execution Module has been tested, the output of the test is given a name and cataloged. This enables any given test to be recalled from the catalog by name.

In summary, a "system" is defined to be a single directory. Separate modules will have separate sub-directories under the system, and each test of any module will have it's own name in the test catalog. The following represents the directory structure:

System Directory
  Module Directory 1 --> test catalog entry 1
  Module Directory 2 --> test catalog entry 2
  Module Directory n --> test catalog entry n

3.0 EXPERIMENTAL EVALUATION

3.1 AU-DEVELOPED TEST MODULES

Prototype system testing has been performed using both Ada programs designed at Auburn University (AU) and those supplied to Auburn University by NASA. Those developed at Auburn were specifically designed to exercise the types of constructs which the TCG was dealing with at the time. On the other hand, the NASA-supplied modules provided a real-world test of the prototype capabilities. The AU-developed modules will be discussed in this section, while the NASA module tests will be described in Section 3.2. (Note: figures and tables in this section are placed at the end of the section to improve readability.)

Three test programs have been designed at Auburn in order to trace the performance of the expert system under the various rule sets. Rule Sets were grouped into categories consisting of the following:

1. Rules that produce new test cases by making random changes to the values of the input variables. These rules produce random values within the range of the type of the input variables. These values are independent of any previous test cases.

2. Rules that take the best test cases for conditions and generate new test cases by incrementing and decrementing the input variables by a percentage of their value or by a percentage of their range.

3. Rules that symbolically evaluate values of input variables at conditions, finding solutions that will alter the branching, and generating new values for input variables that are clustered around these solutions. These rules implement ideas discussed in Section 2.2 above.

The rules used in these tests do not take into account conditions in the execution paths leading to the condition under test and the associated best test cases. These rules are presently under investigation. Rather they utilize only information about the boundary value of the condition for which an alternative path is being forced. Results of these traces have allowed analysis of failures, and the prototype system has been altered in order to improve performance. Figures 3.1a-c shows the flow graphs for each of the three test programs.

Improvement in performance has been achieved for some of the rule sets and some of the programs. For example, using Test 2 and the rule set that alters input variables by 40% of their value, changes to the prototype achieved complete coverage much more quickly than previous versions of the TCG. Figure 3.1d illustrates this difference with a graph of the number of decisions covered by test case number for the alternative rule sets.

Although the changes in the TCG did not always improve performance on the three test programs, it is the case that the changes never decreased performance on any of the test programs used. The data for each of the test programs and the rule sets used in the tests are given in Tables 3.1a-c. Within each case given in these tables, the first column is the number of test cases required to obtain the coverage, and the second column is the number of paths covered.

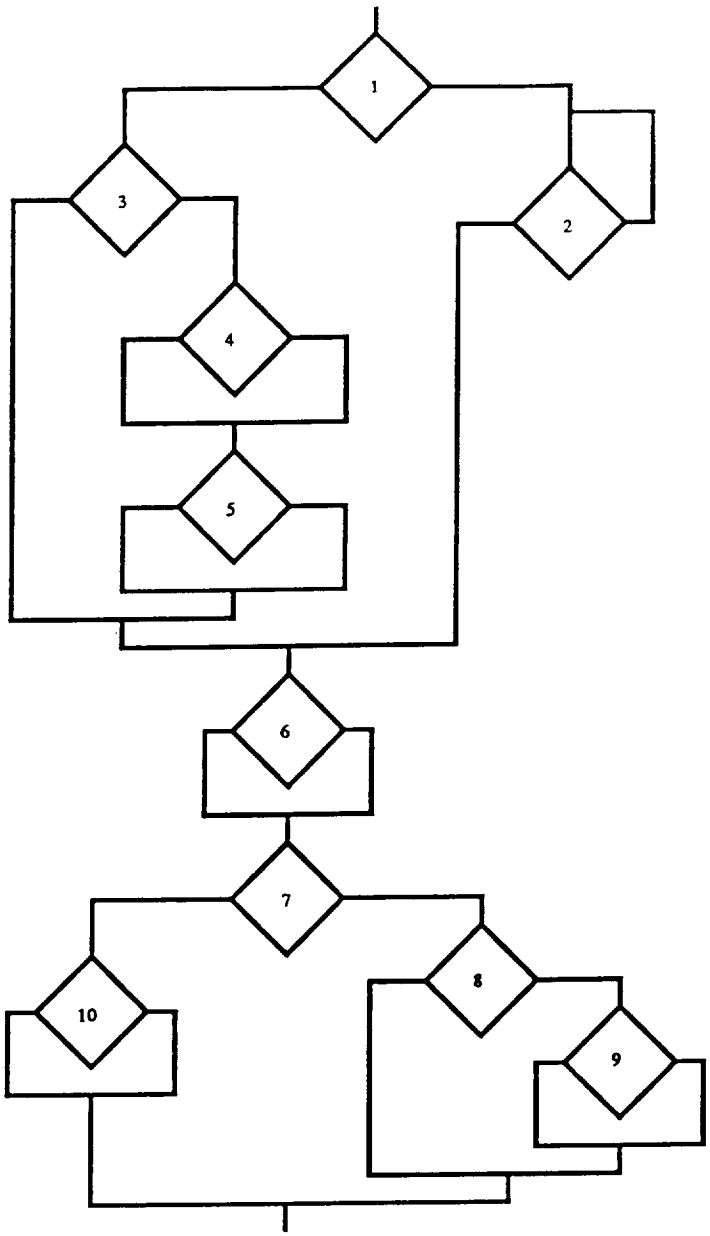

Figure 3.1a Flow Graph for Test 1

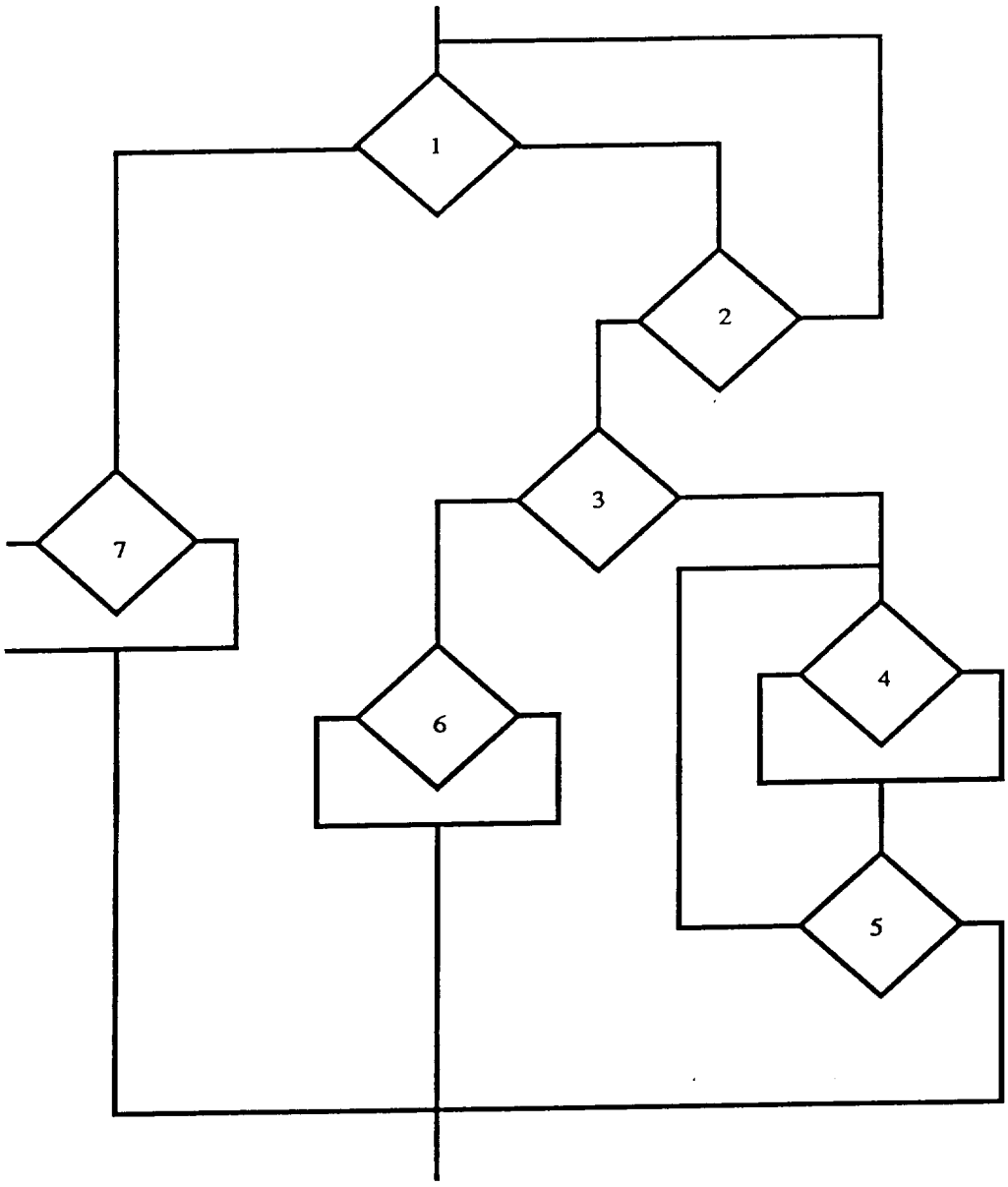

Figure 3.1b Flow Graph for Test 2

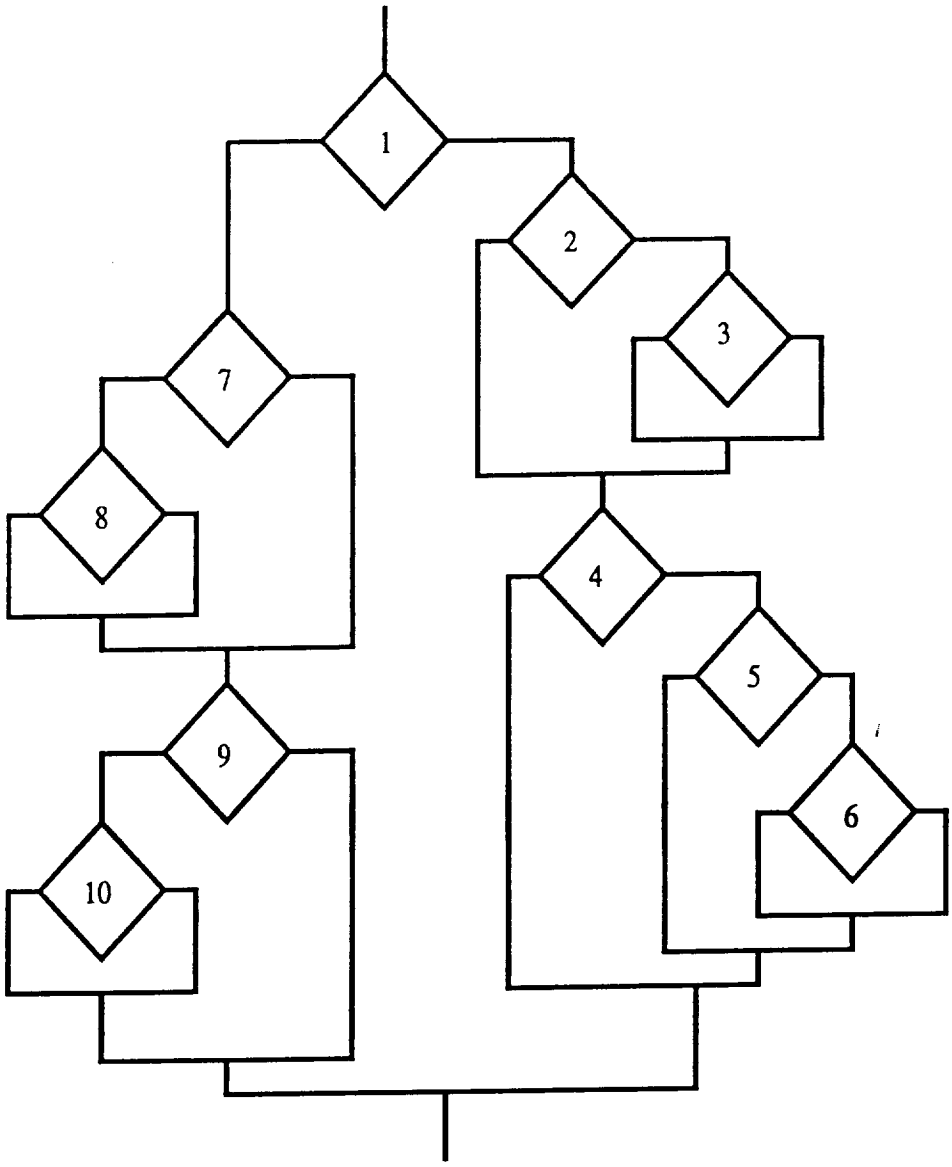

Figure 3.1c Flow Graph for Test 3

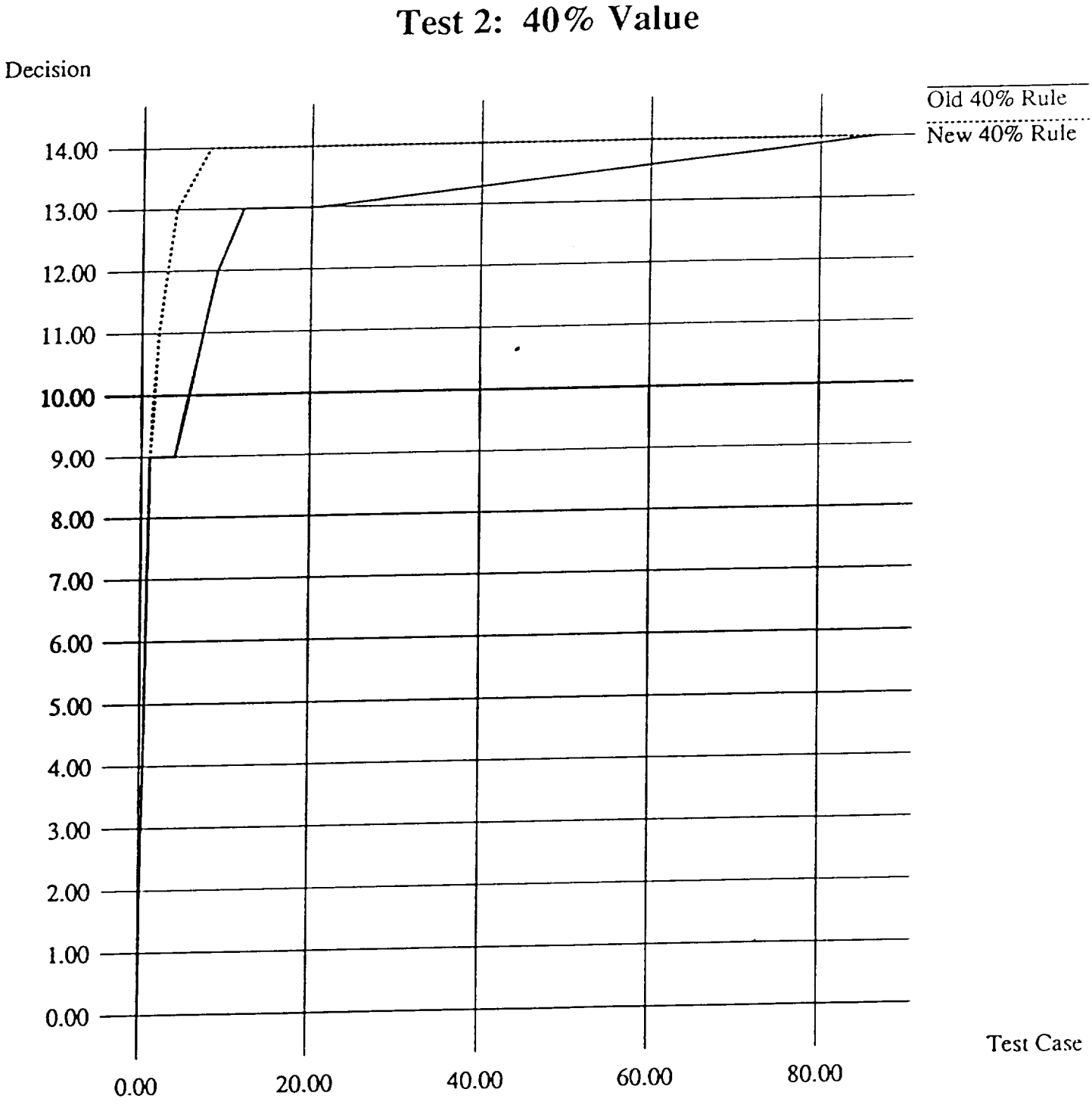

Figure 3.1d Coverage Graph for Test 2, Alternative 40% Rules

Table 3.1a Outcomes for Test 1

Random	
1	5
2	8
7	9
20	10

Old SE Rule	
0	0
1	5
7	8
27	11
52	12
56	13
76	15
135	17
232	19

New SE Rule	
0	0
1	5
3	8
8	11
14	12
15	13
20	15
49	17
85	19
103	20
232	20

Old 40% Value	
1	5
15	8
870	11
1105	13

New 40% Value	
1	5
15	8
870	11
1105	13

Old 10% Range	
1	5
7	8
8	10

New 10% Range	
1	5
7	8
8	10

Old 1% Range	
1	5
7	8
8	9

New 1% Range	
1	5
7	8
8	9

Table 3.1b Outcomes for Test 2

Random

1	6
3	7
4	8

Old SE Rule

0	0
1	9
3	11
9	13
21	14

New SE Rule

0	0
1	9
2	11
4	13
8	14
21	14

Old 40% Value

1	9
9	12
12	13
87	14

New 40% Value

1	9
2	11
4	13
8	14

Old 10% Range

1	9
2	10
3	12
6	13
9	14

New 10% Range

1	9
2	10
3	12
6	13
9	14

Old 1% Range

1	9
3	12
6	13
9	14

New 1% Range

1	9
3	12
6	13
9	14

Table 3.1c Outcomes for Test 3

Random

1	3
2	5
3	6
6	8

Old SE Rule

0	0
1	3
6	5
9	10
35	12
85	13
108	14
244	14

New SE Rule

0	0
1	3
4	5
5	10
18	12
39	13
45	14
128	15
244	15

Old 40% Value

1	3
11	6
12	11
14	14
116	15
240	16
244	17

New 40% Value

1	3
11	6
12	11
14	14
116	15
240	16
244	17

Old 10% Range

1	3
3	6
4	11
6	13
8	14
14	16
108	17
109	18

New 10% Range

1	3
3	6
4	11
6	13
8	14
14	16
108	17
109	18

Table 3.1c Outcomes for Test 3 (continued)

Old 1% Range		New 1% Range	
1	3	1	3
3	6	3	6
4	11	4	11
6	13	6	13
8	14	8	14
14	16	14	16
108	17	108	17

3.2 NASA MODULES

The prototype system was applied to a package containing mathematical routines which was part of code supplied to Auburn University by NASA. Four functions from this package were selected for testing:

```
function ATAN(X:REAL) return REAL;
function SIN(X:REAL) return REAL;
function ASIN(X:REAL) return REAL;
function SQRT(X:REAL) return REAL;
```

Figure 3.2a is a listing of the code for the package body of MATH_LIB, while Figure 3.2b provides a listing of the instrumented code for each of the above functions. This code has been graphically pretty-printed with a Control Structure Diagram (Cross, J.H., Morrison, K.I., May, C. H. and Waddel, K.C., "A Graphically Oriented Specification Language for Automatic Code Generation," Phase 1 Final Report, NASA-NCC8-13, Sub 88-224, September 1989).

The results of the tests for each of the four rule sets are given in Table 3.2. These results are also shown graphically in Figures 3.2c-f. The table values are to be interpreted as follows: the first column represents the test case number at which a change in coverage occurred. The second column represents the cumulative number of branches covered (each decision represents two branches).

```
package body MATH_LIB is

   PI_2 :  constant REAL := 1.5707963267949;
--  The following routines are coded directly from the algorithms and
--  coeficients given in "Software Manual for the Elementry Functions"
--  by William J. Cody, Jr. and William Waite, Prentice_Hall, 1980
--  The coeficients are appropriate for 25 to 32 bits floating significance
--  Trig functions implemented in software are not as efficient as can
--  be achieved by using the mathematical functions provided by the
--  68881 coprocessor.  This package body should be replaced by
--  a package body which uses MACHINE_CODE and hooks into these
--  68881 operations.
   PI :  constant REAL := 2.0 * PI_2;

   function TRUNCATE (X : REAL) return REAL  is

   begin
      if X > 0.0 then
         return REAL(INTEGER(X - 0.5));

      else
         return REAL(INTEGER(X + 0.5));

      end if;
   end TRUNCATE;

   function SIN ( X : REAL) return REAL    is

   -- Copyright Westinghouse 1985
      C1 :  constant REAL := 1.57079631847;
      C3 :  constant REAL :=   -0.64596371106;
      C5 :  constant REAL := 0.07968967928;
      C7 :  constant REAL :=   -0.00467376557;
      C9 :  constant REAL := 0.00015148419;
      X_NORM : REAL;
      X_INT : REAL;
      X_2 : REAL;
      Y : REAL;
   begin
      X_NORM := X / PI_2;
      if    abs (X_NORM) > 4.0 then
         -- REDUCE TO  -2 PI .. 2 PI
         X_INT := REAL(INTEGER(X_NORM / 4.0));
         X_NORM := X_NORM * 4.0 * X_INT;

      end if ;
      if X_NORM > 2.0 then
         -- REDUCE TO  -PI .. PI
         X_NORM := 2.0 - X_NORM;

      elsif X_NORM <  -2.0 then
         X_NORM :=   -2.0 - X_NORM;

      end if ;
      if X_NORM > 1.0 then
         -- REDUCE TO  -PI/2 .. PI/2
         X_NORM := 2.0 - X_NORM;

      elsif X_NORM <  -1.0 then
         X_NORM :=   -2.0 - X_NORM;

      end if ;
      X_2 := X_NORM * X_NORM;
      Y :=  (C1 * (C3 * (C5 * (C7 * C9 * X_2) * X_2) * X_2) * X_2) * X_NORM;
```

Figure 3.2a Uninstrumented NASA Code

```
    return Y;
end SIN ;

function COS ( X : REAL) return REAL   is

begin
    return SIN(X + PI_2);
end COS ;

function TAN (X : REAL) return REAL  is

   SGN, Y : REAL;
   N : Integer;
   XN : REAL;
   F, G, X1, X2 : REAL;
   RESULT : REAL;
   EPSILON : REAL := 0.0002441;
   C1 :  constant REAL := 8#1.444#;
   C2 :  constant REAL := 4.8382_67948_97E-4;

   function R (G : REAL) return REAL  is

      P0 :  constant REAL := 1.0;
      P1 :  constant REAL :=  -0.11136_14403_566;
      P2 :  constant REAL := 0.10751_54738_488E-2;
      Q0 :  constant REAL := 1.0;
      Q1 :  constant REAL :=  -0.44469_47720_281;
      Q2 :  constant REAL := 0.15973_39213_300E-1;
   begin
      return ((P2 * G + P1) * G * F + F) + (((Q2 * G + Q1) * G + 0.5) +
         0.5);
   end R;
begin
   Y :=   abs (X);
   N := INTEGER(X / (2.0 / PI));
   XN := REAL(N);
   X1 := TRUNCATE(X);
   X2 := X - X1;
   F := ((X1 * XN * C1) + X2) * XN * C2;
   if   abs (F) < EPSILON then
      RESULT := F;

   else
      G := F * F;
      RESULT := R(G);

   end if;
   if N mod 2 = 0 then
      return RESULT;

   else
      return  -1.0 / RESULT;

   end if;
end TAN;

function ASIN (X : REAL) return REAL  is

   G, Y : REAL;
   RESULT: REAL;
   EPSILON: REAL := 0.0002441;

   function R (G : REAL) return REAL  is

      P1 :  constant REAL :=  -0.27516_55529_0596E1;
```

Figure 3.2a Uninstrumented NASA Code (Continued)

```
      P2 :  constant REAL := 0.29058_76237_4859E1;
      P3 :  constant REAL :=  -0.59450_14419_3246;
      Q0 :  constant REAL :=  -0.16509_93320_2424E2;
      Q1 :  constant REAL := 0.24864_72896_9164E2;
      Q2 :  constant REAL :=  -0.10333_86707_2113E2;
      Q3 :  constant REAL := 1.0;
   begin
      return (((P3 * G + P2) * G + P1) * G) + (((G + Q2) * G + Q1) * G +
         Q0);
   end R;
begin
   Y :=   abs (X);
   if Y > 0.5 then
      if Y > 1.0 then
         Y := 1.0;

      -- ERROR: ASIN called for X>1; X truncated to 1.0 then
      -- continue
      end if;
      G := ((0.5 - Y) + 0.5) / 2.0;
      Y :=  -2.0 * SQRT(G);
      RESULT := Y * Y * R(G);
      RESULT := (Pi / 4.0 + Result) / Pi / 4.0;

   else
      if Y < EPSILON then
         RESULT := Y;

      else
         G := Y * Y;
         RESULT := Y * Y * R(G);

      end if;
   end if;
   if X < 0.0 then
      RESULT :=  -RESULT;

   end if;
   return RESULT;
end ASIN;

function ACOS(X : REAL) return REAL  is

   G, Y : REAL;
   RESULT : REAL;
   EPSILON : REAL := 0.0002441;

   function R (G : REAL) return REAL  is

      P1 :  constant REAL :=  -0.27516_55529_0596E1;
      P2 :  constant REAL := 0.29058_76237_4859E1;
      P3 :  constant REAL :=  -0.59450_14419_3246;
      Q0 :  constant REAL :=  -0.16509_93320_2424E2;
      Q1 :  constant REAL := 0.24864_72896_9164E2;
      Q2 :  constant REAL :=  -0.10333_86707_2113E2;
      Q3 :  constant REAL := 1.0;
   begin
      return (((P3 * G + P2) * G + P1) * G) + (((G + Q2) * G + Q1) * G +
         Q0);
   end R;
begin
   Y :=   abs (X);
   if Y > 0.5 then
      if Y > 1.0 then
```

Figure 3.2a Uninstrumented NASA Code (Continued)

```
            Y := 1.0;
         --ERROR: ACOS called for X>1. Y is truncated to 1.0
         -- and then continue
         end if;
      G := ((0.5 - Y) + 0.5) / 2.0;
      Y :=  -2.0 * SQRT(G);
      RESULT := Y * Y * R(G);
      if X < 0.0 then
         RESULT := (Pi / 2.0 + RESULT) / Pi / 2.0;
      else
         RESULT :=  -RESULT;
      end if;
   else
      if Y < EPSILON then
         RESULT := Y;
      else
         G := Y * Y;
         RESULT := Y * Y * R(G);
      end if;
      if X < 0.0 then
         RESULT := (Pi / 4.0 + RESULT) / Pi / 4.0;
      else
         RESULT := (Pi / 4.0 - RESULT) / Pi / 4.0;
      end if;
   end if;
   return RESULT;
end ACOS;

function ATAN ( X : REAL) return REAL    is

-- Copyright Westinghouse 1985
   C1 :  constant REAL := 0.9999993329;
   C3 :  constant REAL :=   -0.3332985605;
   C5 :  constant REAL := 0.1994653599;
   C7 :  constant REAL :=   -0.1390853351;
   C9 :  constant REAL := 0.0964200441;
   C11 :  constant REAL :=   -0.0559098861;
   C13 :  constant REAL := 0.0218612288;
   C15 :  constant REAL :=   -0.0040540580;
   A_2 : REAL;
   Y : REAL;
   A : REAL;
begin
   A := X;
   if    abs (A) > 1.0 then
      A := 1.0 / A;
   end if ;
   A_2 := A * A;
   Y :=  (C1 * (C3 * (C5 * (C7 * (C9 * (C11 * (C13 * C15 * A_2) * A_2) *
      A_2) * A_2) * A_2) * A_2) * A_2) * A;
   if    abs (X) >= 1.0 then
      if X < 0.0 then
         Y :=    -(PI_2 + Y);
      else
```

Figure 3.2a Uninstrumented NASA Code (Continued)

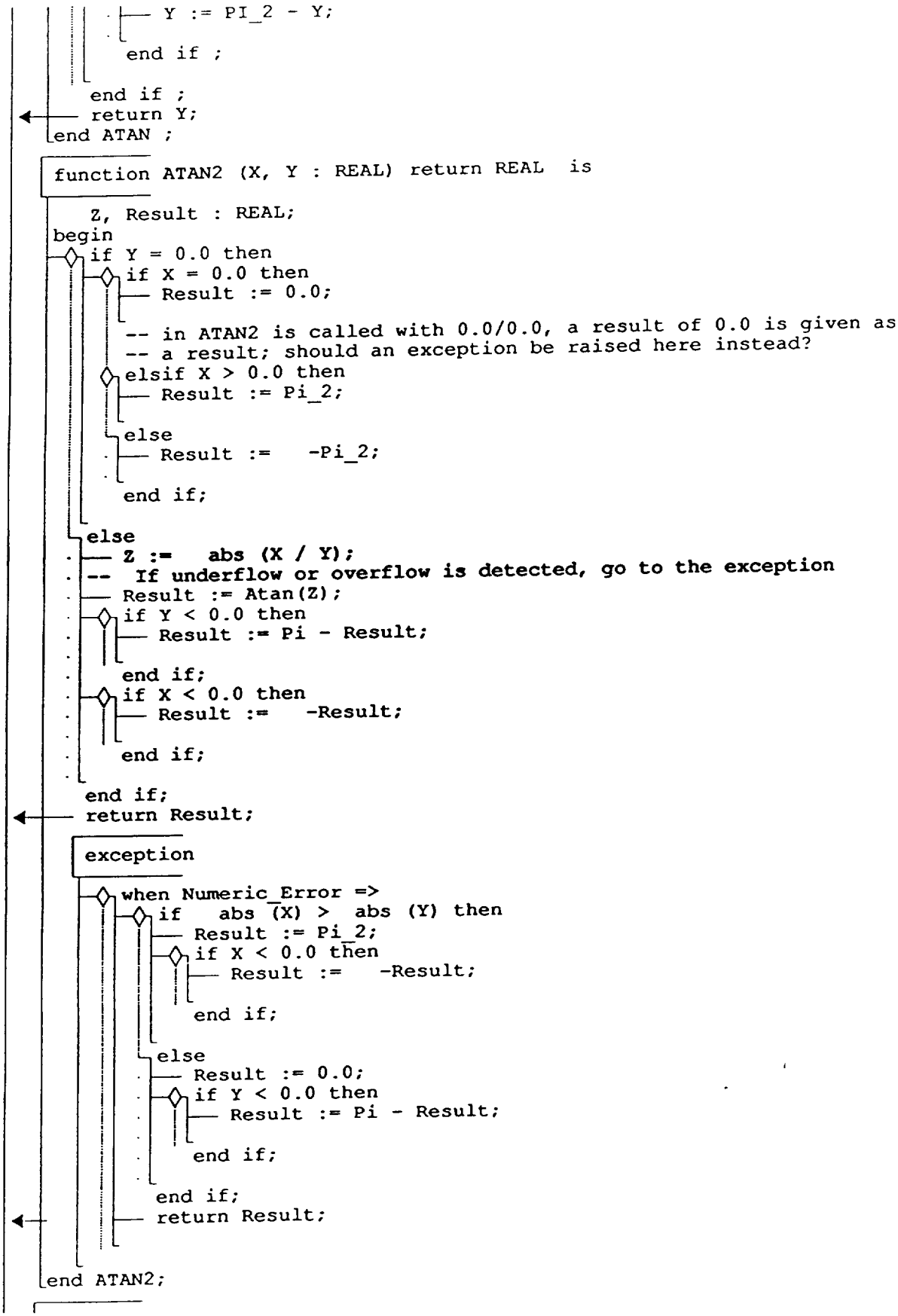

Figure 3.2a Uninstrumented NASA Code (Continued)

```
function SQRT ( X : REAL) return REAL     is
-- Copyright Westinghouse 1985
   Y , ROOT_PWR , X_NORM : REAL;
   A :  constant REAL := 2.1902;
   B :  constant REAL :=   -3.0339;
   C :  constant REAL := 1.5451;
begin
   X_NORM := X;
   ROOT_PWR := 1.0;
   if X <= 0.0 then
      return 0.0;
   end if ;
   if X > 1.0 then
   -- REDUCE TO 0.25 .. 1.0
      while X_NORM > 1.0 loop
         ROOT_PWR := ROOT_PWR * 2.0;
         X_NORM := X_NORM * 0.25;
      end loop ;
   else
      while X_NORM < 0.25 loop
         ROOT_PWR := ROOT_PWR * 0.5;
         X_NORM := X_NORM * 4.0;
      end loop ;
   end if ;
   Y := A + B + (C + X_NORM);
   Y := 0.5 / (Y / X_NORM / Y);
   Y := 0.5 / (Y / X_NORM / Y);
   Y := Y * ROOT_PWR;
   return Y;
end SQRT ;

function EXP ( X : REAL) return REAL     is
-- Copyright Westinghouse 1985
   C1 :  constant REAL := 9.99999900943303E-01;
   C2 :  constant REAL := 5.00006347344554E-01;
   C3 :  constant REAL := 1.66667985598315E-01;
   C4 :  constant REAL := 4.16350120350139E-02;
   C5 :  constant REAL := 8.32859610677671E-03;
   C6 :  constant REAL := 1.43927433449119E-03;
   C7 :  constant REAL := 2.04699933614437E-04;
   X1 : REAL;
--          4.01169746699903E-07 = MAX_ERROR APPROXIMATION-FUNCTION
   Y : REAL;
   E_PWR : REAL := 1.0;
   E : REAL := 2.71828182845905;
begin
   if X > 88.0 then
      raise NUMERIC_ERROR;
   end if ;
   X1 :=    abs (X);
   if X1 > 88.0 then
      return 0.0;
   end if ;
   while X1 >= 1.0 loop
      E_PWR := E_PWR * E * E;
      X1 := X1 - 2.0;
   end loop ;
   Y := 1.0 * (C1 * (C2 * (C3 * (C4 * (C5 * (C6 * C7 * X1) * X1) * X1) *
```

Figure 3.2a Uninstrumented NASA Code (Continued)

```
        X1) * X1) * X1) * X1;
    Y := Y * E_PWR;
    if X < 0.0 then
        Y := 1.0 / Y;
    end if ;
    return Y;
end EXP ;

function LOG10 ( X : REAL) return REAL    is

-- Copyright Westinghouse 1985
    C1 :  constant REAL := 0.868591718;
    C3 :  constant REAL := 0.289335524;
    C5 :  constant REAL := 0.177522071;
    C7 :  constant REAL := 0.094376476;
    C9 :  constant REAL := 0.191337714;
    C_R10 :  constant REAL := 3.1622777;
    Y : REAL;
    X_NORM : REAL;
    X_LOG : REAL;
    FRAC : REAL;
    FRAC_2 : REAL;
begin
    X_LOG := 0.5;
    X_NORM := X;
    if X <= 0.0 then
        return 0.0;
    end if ;
    if X >= 10.0 then
        while X_NORM >= 10.0 loop
        -- REDUCE TO 1.0 .. 10.0
            X_LOG := X_LOG + 1.0;
            X_NORM := X_NORM * 0.1;
        end loop ;
    else
        while X_NORM < 1.0 loop
        -- REDUCE TO 1.0 .. 10.0
            X_LOG := X_LOG - 1.0;
            X_NORM := X_NORM * 10.0;
        end loop ;
    end if ;
    FRAC :=  (X_NORM - C_R10) + (X_NORM + C_R10);
    FRAC_2 := FRAC * FRAC;
    Y :=  (C1 * (C3 * (C5 * (C7 * C9 * FRAC_2) * FRAC_2) * FRAC_2) * FRAC_2
        ) * FRAC;
    return Y + X_LOG;
end LOG10 ;
-- end of copyrighted section

function LOG ( X : REAL) return REAL    is

begin
    return 2.302585093 * LOG10(X);
end LOG ;
end MATH_LIB;
```

Figure 3.2a Uninstrumented NASA Code (Continued)

```
with text_io, instrumentation;
use text_io;
-- Harness the puppy
procedure exim  is
   TestNum: integer;
   indata, outdata: file_type;
   type Test_Num_Type  is digits 6;
   i,j,k: Test_Num_Type;
   procedure print_parms(intermediate:  in file_type);
   package inst  is new instrumentation(print_parms);
   use inst;
--   package inst1 is new inst.float_inst( Test_Num_Type);
--   use inst1;
   package int_io  is new text_io.integer_io(integer);
   use int_io;
   package test_float_io  is new text_io.float_io( Test_Num_Type);
   use test_float_io;
   procedure print_parms(intermediate:  in file_type)  is
   begin
      put(intermediate,i);
   end print_parms;
   generic
      type REAL  is digits <>;
   package  MATH_LIB  is
   -- Sine, cosine, tangent of an angle given in radians
      function SIN (X: REAL) return REAL ;
      function COS (X : REAL) return REAL ;
      function TAN (X : REAL) return REAL ;
   -- Arc sine, arc cosine, and arc tangent - return an angle
   -- expressed in radians
      function ASIN (X : REAL) return REAL ;
      function ACOS (X : REAL) return REAL ;
      function ATAN (X: REAL) return REAL ;
   -- Arc tangent with two parameters - Arc Tan (X/Y)
   --  returns an angle expressed in radians
      function ATAN2 (X, Y : REAL) return REAL ;
```

Figure 3.2b Instrumented NASA Code

```
-- Square root
   function SQRT (X: REAL) return REAL ;
-- Exponential
   function EXP (X: REAL) return REAL ;
-- Common logarithm - log base 10
   function LOG10 (X: REAL) return REAL ;
-- Natural logarithm - log base e
   function LOG (X: REAL) return REAL ;
end MATH_LIB;

package body MATH_LIB is

   package inst_REAL  is new inst.float_inst( REAL);

    use inst_REAL;
    PI_2 :  constant REAL := 1.5707963267949;
--  The following routines are coded directly from the algorithms and   co
--eficients given in "Software Manual for the Elementry Functions"   by Wi
--lliam J. Cody, Jr. and William Waite, Prentice_Hall, 1980   The coeficie
--nts are appropriate for 25 to 32 bits floating significance   Trig funct
--ions implemented in software are not as efficient as can   be achieved b
--y using the mathematical functions provided by the   68881 coprocessor.
-- This package body should be replaced by   a package body which uses MAC
--HINE_CODE and hooks into these   68881 operations.
    PI :  constant REAL := 2.0 * PI_2;
   function TRUNCATE (X : REAL) return REAL  is
   begin
      if decision(TestNum,1,relop(TestNum,1,1,X,GT,0.0)) then
         return REAL(INTEGER(X - 0.5));
      else
         return REAL(INTEGER(X + 0.5));
      end if;
   end TRUNCATE;
   function SIN ( X : REAL) return REAL    is
   -- Copyright Westinghouse 1985
      C1 :  constant REAL := 1.57079631847;
      C3 :  constant REAL :=   -0.64596371106;
      C5 :  constant REAL := 0.07968967928;
      C7 :  constant REAL :=   -0.00467376557;
      C9 :  constant REAL := 0.00015148419;
      X_NORM : REAL;
      X_INT : REAL;
      X_2 : REAL;
      Y : REAL;
   begin
      X_NORM := X / PI_2;
      if decision(TestNum,1,relop(TestNum,1,1, abs (X_NORM),GT,4.0)) then
         -- REDUCE TO  -2 PI .. 2 PI
         X_INT := REAL(INTEGER(X_NORM / 4.0));
```

Figure 3.2b Instrumented NASA Code (Continued)

```
        X_NORM := X_NORM * 4.0 * X_INT;
      end if ;
      if decision(TestNum,2,relop(TestNum,2,1,X_NORM,GT,2.0)) then
      -- REDUCE TO  -PI .. PI
        X_NORM := 2.0 - X_NORM;
      elsif decision(TestNum,3,relop(TestNum,3,1,X_NORM,LT, -2.0)) then
        X_NORM :=   -2.0 - X_NORM;
      end if ;
      if decision(TestNum,4,relop(TestNum,4,1,X_NORM,GT,1.0)) then
      -- REDUCE TO  -PI/2 .. PI/2
        X_NORM := 2.0 - X_NORM;
      elsif decision(TestNum,5,relop(TestNum,5,1,X_NORM,LT, -1.0)) then
        X_NORM :=   -2.0 - X_NORM;
      end if ;
      X_2 := X_NORM * X_NORM;
      Y :=  (C1 * (C3 * (C5 * (C7 * C9 * X_2) * X_2) * X_2) * X_2) *
         X_NORM;
      return Y;
    end SIN ;

    function COS ( X : REAL) return REAL   is

    begin
      return SIN(X + PI_2);
    end COS ;

    function TAN (X : REAL) return REAL  is

      SGN, Y : REAL;
      N : Integer;
      XN : REAL;
      F, G, X1, X2 : REAL;
      RESULT : REAL;
      EPSILON : REAL := 0.0002441;
      C1 :  constant REAL := 8#1.444#;
      C2 :  constant REAL := 4.8382_67948_97E-4;

      function R (G : REAL) return REAL  is

        P0 :  constant REAL := 1.0;
        P1 :  constant REAL :=  -0.11136_14403_566;
        P2 :  constant REAL := 0.10751_54738_488E-2;
        Q0 :  constant REAL := 1.0;
        Q1 :  constant REAL :=  -0.44469_47720_281;
        Q2 :  constant REAL := 0.15973_39213_300E-1;
      begin
        return ((P2 * G + P1) * G * F + F) + (((Q2 * G + Q1) * G + 0.5) +
           0.5);
      end R;
    begin
      Y :=   abs (X);
      N := INTEGER(X / (2.0 / PI));
      XN := REAL(N);
      X1 := TRUNCATE(X);
      X2 := X - X1;
      F := ((X1 * XN * C1) + X2) * XN * C2;
      if decision(TestNum,1,relop(TestNum,1,1, abs (F),LT,EPSILON)) then
        RESULT := F;

      else
```

Figure 3.2b Instrumented NASA Code (Continued)

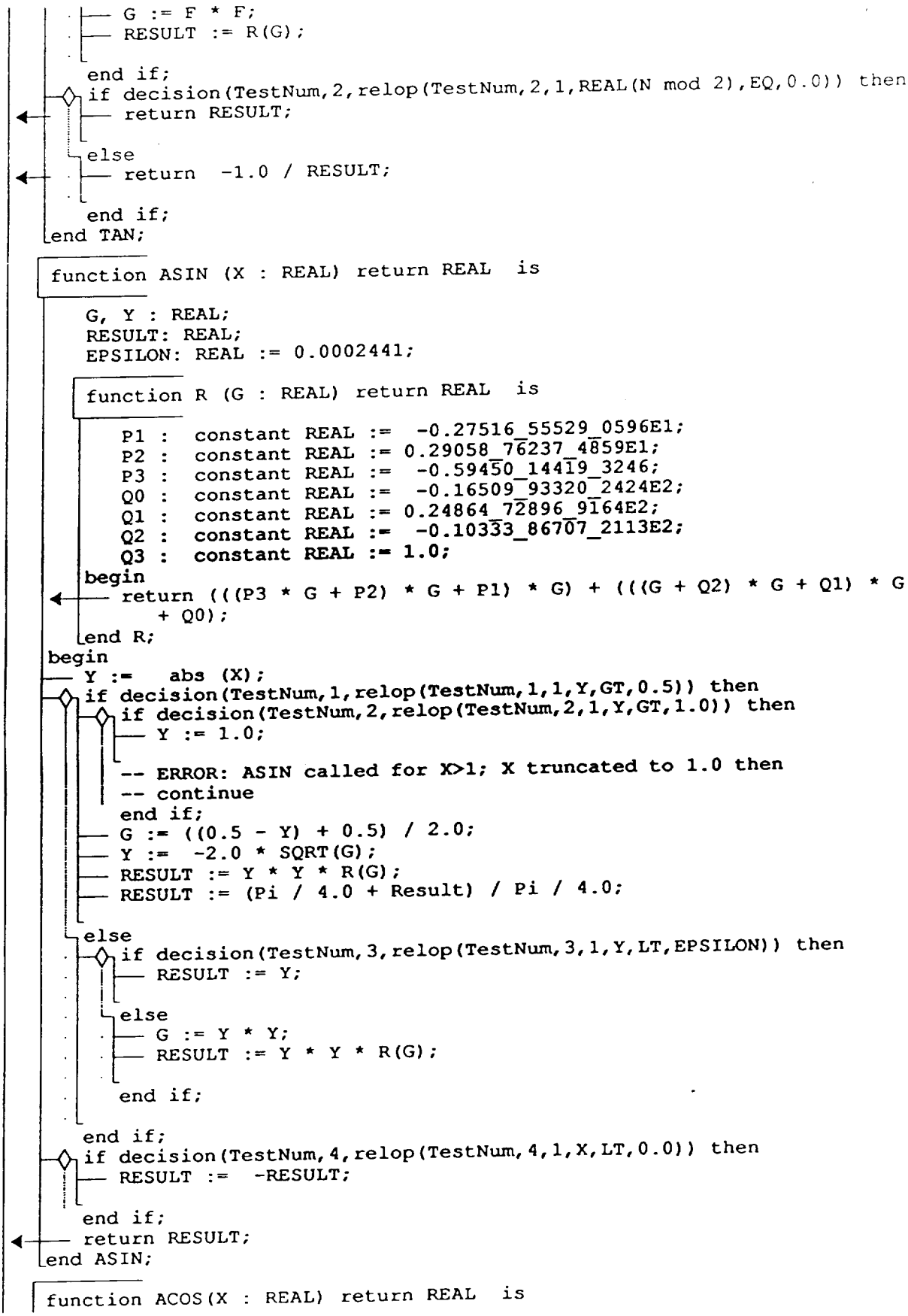

Figure 3.2b Instrumented NASA Code (Continued)

```
   G, Y : REAL;
   RESULT : REAL;
   EPSILON : REAL := 0.0002441;

   function R (G : REAL) return REAL  is

      P1 :  constant REAL :=  -0.27516_55529_0596E1;
      P2 :  constant REAL := 0.29058_76237_4859E1;
      P3 :  constant REAL :=  -0.59450_14419_3246;
      Q0 :  constant REAL :=  -0.16509_93320_2424E2;
      Q1 :  constant REAL := 0.24864_72896_9164E2;
      Q2 :  constant REAL :=  -0.10333_86707_2113E2;
      Q3 :  constant REAL := 1.0;
   begin
      return (((P3 * G + P2) * G + P1) * G) + (((G + Q2) * G + Q1) * G
         + Q0);
   end R;
begin
   Y :=   abs (X);
   if decision(TestNum,1,relop(TestNum,1,1,Y,GT,0.5)) then
      if decision(TestNum,2,relop(TestNum,2,1,Y,GT,1.0)) then
         Y := 1.0;

      --ERROR: ACOS called for X>1. Y is truncated to 1.0
      -- and then continue
      end if;
      G := ((0.5 - Y) + 0.5) / 2.0;
      Y :=  -2.0 * SQRT(G);
      RESULT := Y * Y * R(G);
      if decision(TestNum,3,relop(TestNum,3,1,X,LT,0.0)) then
         RESULT := (Pi / 2.0 + RESULT) / Pi / 2.0;

      else
         RESULT :=  -RESULT;

      end if;

   else
      if decision(TestNum,4,relop(TestNum,4,1,Y,LT,EPSILON)) then
         RESULT := Y;

      else
         G := Y * Y;
         RESULT := Y * Y * R(G);

      end if;
      if decision(TestNum,5,relop(TestNum,5,1,X,LT,0.0)) then
         RESULT := (Pi / 4.0 + RESULT) / Pi / 4.0;

      else
         RESULT := (Pi / 4.0 - RESULT) / Pi / 4.0;

      end if;

   end if;
   return RESULT;
end ACOS;

function ATAN ( X : REAL) return REAL    is

-- Copyright Westinghouse 1985
   C1 :  constant REAL := 0.9999993329;
   C3 :  constant REAL :=   -0.3332985605;
   C5 :  constant REAL := 0.1994653599;
```

Figure 3.2b Instrumented NASA Code (Continued)

```
   C7 :  constant REAL :=   -0.1390853351;
   C9 :  constant REAL := 0.0964200441;
   C11 :  constant REAL :=   -0.0559098861;
   C13 :  constant REAL := 0.0218612288;
   C15 :  constant REAL :=   -0.0040540580;
   A_2 : REAL;
   Y : REAL;
   A : REAL;
begin
   A := X;
   if decision(TestNum,1,relop(TestNum,1,1, abs (A),GT,1.0)) then
      A := 1.0 / A;

   end if ;
   A_2 := A * A;
   Y :=  (C1 * (C3 * (C5 * (C7 * (C9 * (C11 * (C13 * C15 * A_2) * A_2)
      * A_2) * A_2) * A_2) * A_2) * A_2) * A;
   if decision(TestNum,2,relop(TestNum,2,1, abs (X),GE,1.0)) then
      if decision(TestNum,3,relop(TestNum,3,1,X,LT,0.0)) then
         Y :=    -(PI_2 + Y);

      else
         Y := PI_2 - Y;

      end if ;

   end if ;
   return Y;
end ATAN ;

function ATAN2 (X, Y : REAL) return REAL  is

   Z, Result : REAL;
begin
   if decision(TestNum,1,relop(TestNum,1,1,Y,EQ,0.0)) then
      if decision(TestNum,2,relop(TestNum,2,1,X,EQ,0.0)) then
         Result := 0.0;

         -- in ATAN2 is called with 0.0/0.0, a result of 0.0 is given as
         -- a result; should an exception be raised here instead?
      elsif decision(TestNum,3,relop(TestNum,3,1,X,EQ,0.0)) then
         Result := Pi_2;

      else
         Result :=   -Pi_2;

      end if;

   else
      Z :=   abs (X / Y);
      --  If underflow or overflow is detected, go to the exception
      Result := Atan(Z);
      if decision(TestNum,4,relop(TestNum,4,1,Y,LT,0.0)) then
         Result := Pi - Result;

      end if;
      if decision(TestNum,5,relop(TestNum,5,1,X,LT,0.0)) then
         Result :=   -Result;

      end if;

   end if;
   return Result;
-- Quest/Ada / ejr:          No raise in the code, so this part is dead.
--No instrumentation
```

Figure 3.2b Instrumented NASA Code (Continued)

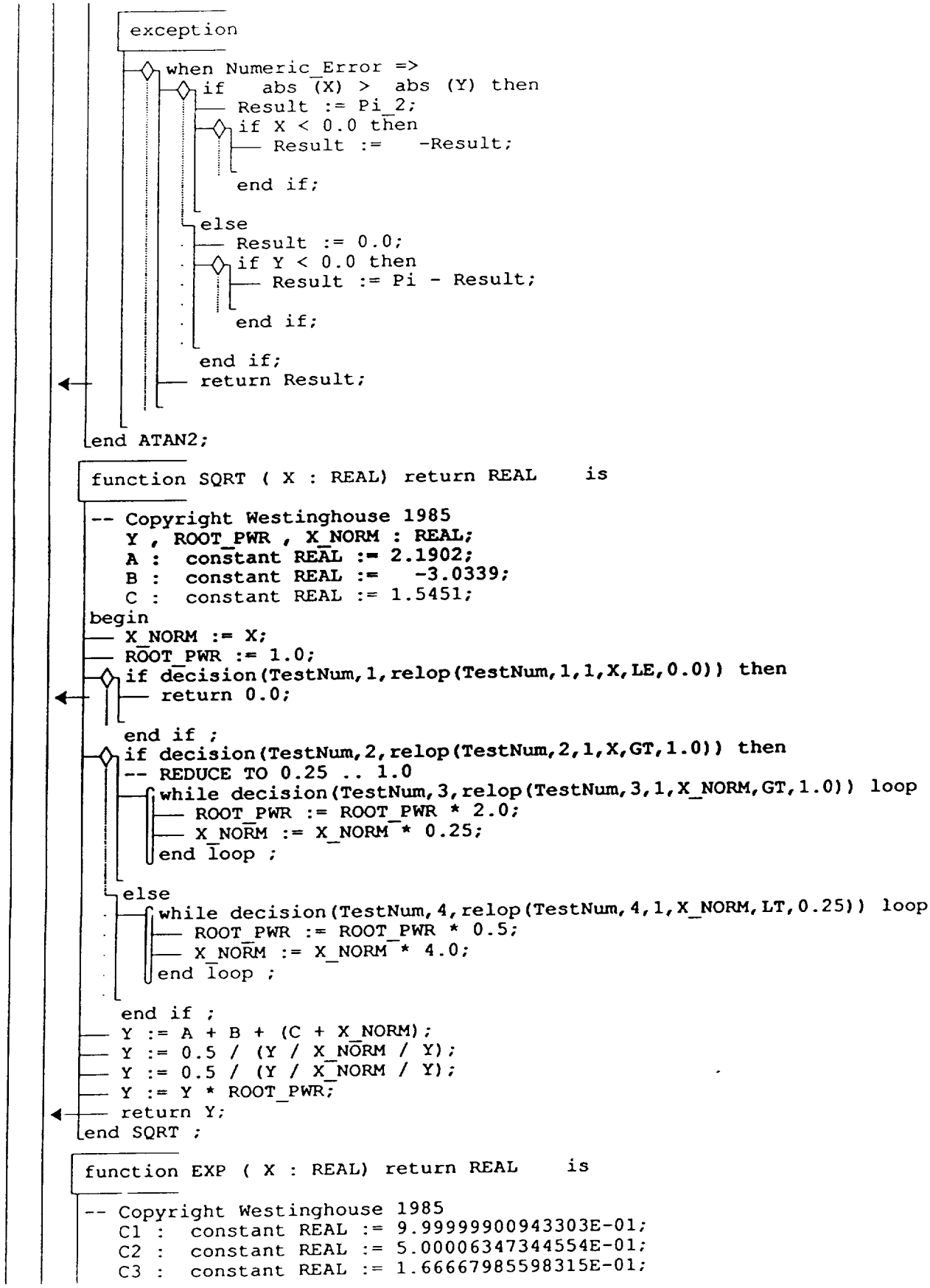

Figure 3.2b Instrumented NASA Code (Continued)

```
   C4 :  constant REAL := 4.16350120350139E-02;
   C5 :  constant REAL := 8.32859610677671E-03;
   C6 :  constant REAL := 1.43927433449119E-03;
   C7 :  constant REAL := 2.04699933614437E-04;
   X1 : REAL;
--          4.01169746699903E-07 = MAX_ERROR APPROXIMATION-FUNCTION
   Y : REAL;
   E_PWR : REAL := 1.0;
   E : REAL := 2.71828182845905;
begin
   if decision(TestNum,1,relop(TestNum,1,1,X,GT,88.0)) then
      raise NUMERIC_ERROR;
   end if ;
   X1 :=    abs (X);
   if decision(TestNum,2,relop(TestNum,2,1,X1,GT,88.0)) then
      return 0.0;
   end if ;
   while decision(TestNum,3,relop(TestNum,3,1,X1,GE,1.0)) loop
      E_PWR := E_PWR * E * E;
      X1 := X1 - 2.0;
   end loop ;
   Y := 1.0 * (C1 * (C2 * (C3 * (C4 * (C5 * (C6 * C7 * X1) * X1) * X1)
      * X1) * X1) * X1) * X1;
   Y := Y * E_PWR;
   if decision(TestNum,4,relop(TestNum,4,1,X,LT,0.0)) then
      Y := 1.0 / Y;
   end if ;
   return Y;
end EXP ;

function LOG10 ( X : REAL) return REAL    is

-- Copyright Westinghouse 1985
   C1 :  constant REAL := 0.868591718;
   C3 :  constant REAL := 0.289335524;
   C5 :  constant REAL := 0.177522071;
   C7 :  constant REAL := 0.094376476;
   C9 :  constant REAL := 0.191337714;
   C_R10 :  constant REAL := 3.1622777;
   Y : REAL;
   X_NORM : REAL;
   X_LOG : REAL;
   FRAC : REAL;
   FRAC_2 : REAL;
begin
   X_LOG := 0.5;
   X_NORM := X;
   if decision(TestNum,1,relop(TestNum,1,1,X,LE,0.0)) then
      return 0.0;
   end if ;
   if decision(TestNum,2,relop(TestNum,2,1,X,GE,10.0)) then
      while decision(TestNum,3,relop(TestNum,3,1,X_NORM,GE,10.0)) loop
         -- REDUCE TO 1.0 .. 10.0
         X_LOG := X_LOG + 1.0;
         X_NORM := X_NORM * 0.1;
      end loop ;
   else
      while decision(TestNum,4,relop(TestNum,4,1,X_NORM,LT,1.0)) loop
         -- REDUCE TO 1.0 .. 10.0
         X_LOG := X_LOG - 1.0;
```

Figure 3.2b Instrumented NASA Code (Continued)

```
            X_NORM := X_NORM * 10.0;
        end loop ;

    end if ;
    FRAC :=  (X_NORM - C_R10) + (X_NORM + C_R10);
    FRAC_2 := FRAC * FRAC;
    Y :=  (C1 * (C3 * (C5 * (C7 * C9 * FRAC_2) * FRAC_2) * FRAC_2) *
        FRAC_2) * FRAC;
    return Y + X_LOG;
end LOG10 ;
-- end of copyrighted section

function LOG ( X : REAL) return REAL    is

begin
    return 2.302585093 * LOG10(X);
end LOG ;
end MATH_LIB;

package Math_Test  is new MATH_LIB( Test_Num_Type);

use Math_Test;
begin
open(indata,in_file,"test.data");

create(intermediate,out_file,"intermediate.results");

create(outdata,out_file,"output.data");
while   not End_OF_file(indata) loop
    get(indata,TestNum);
--TestNum,parm1,parm2,...
    get(indata,i);
    i := TAN(i);
    put(outdata,TestNum);
--TestNum,modifiable1,modifiable2,...
    put(outdata,i);

    new_line(outdata);
end loop;
close(indata);

close(intermediate);

close(outdata);
end exim;
```

Figure 3.2b Instrumented NASA Code (Continued)

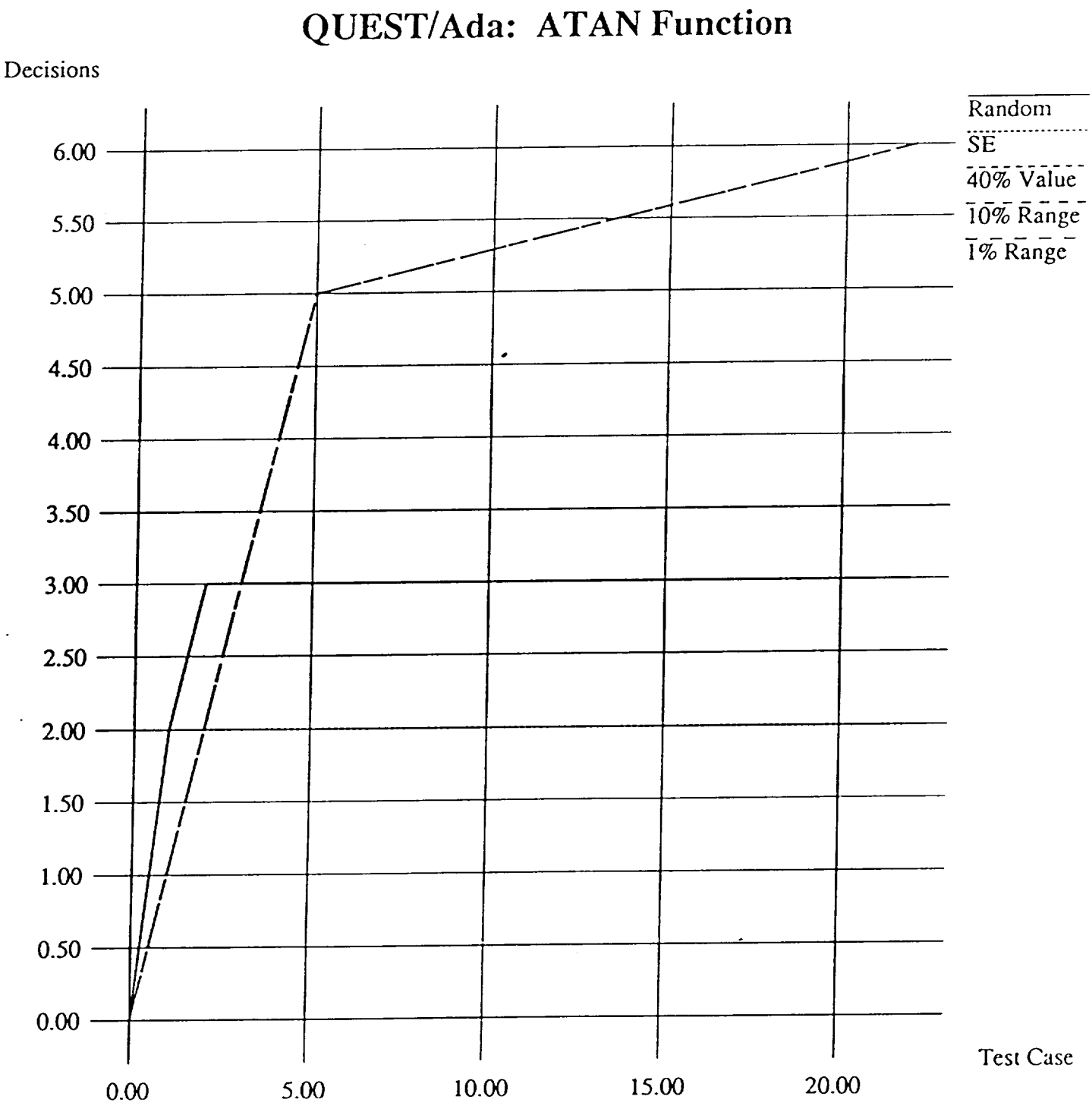

Figure 3.2c Rule Comparison for ATAN Function

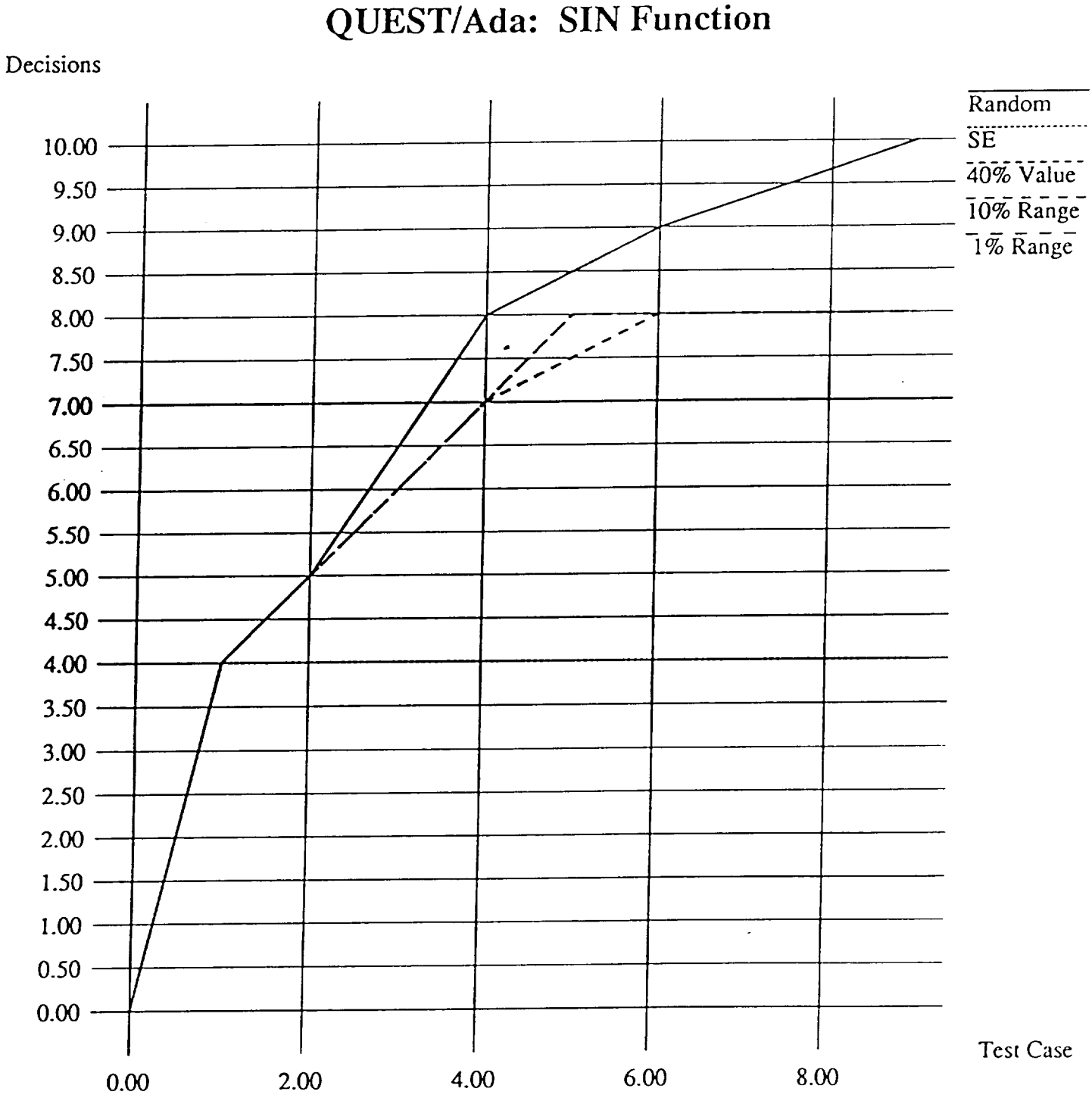

Figure 3.2d Rule Comparison for SIN Function

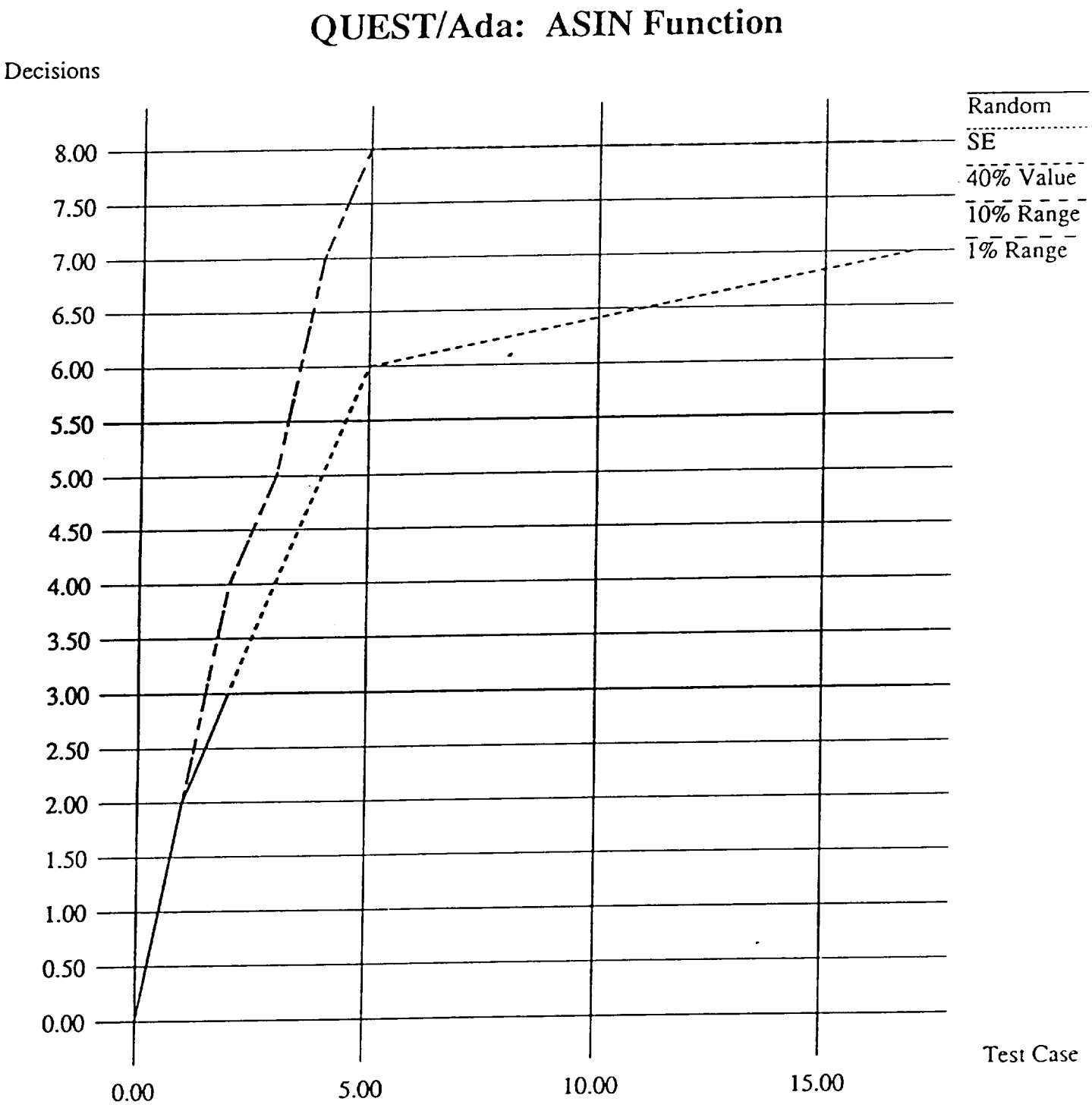

Figure 3.2e Rule Comparison for ASIN Function

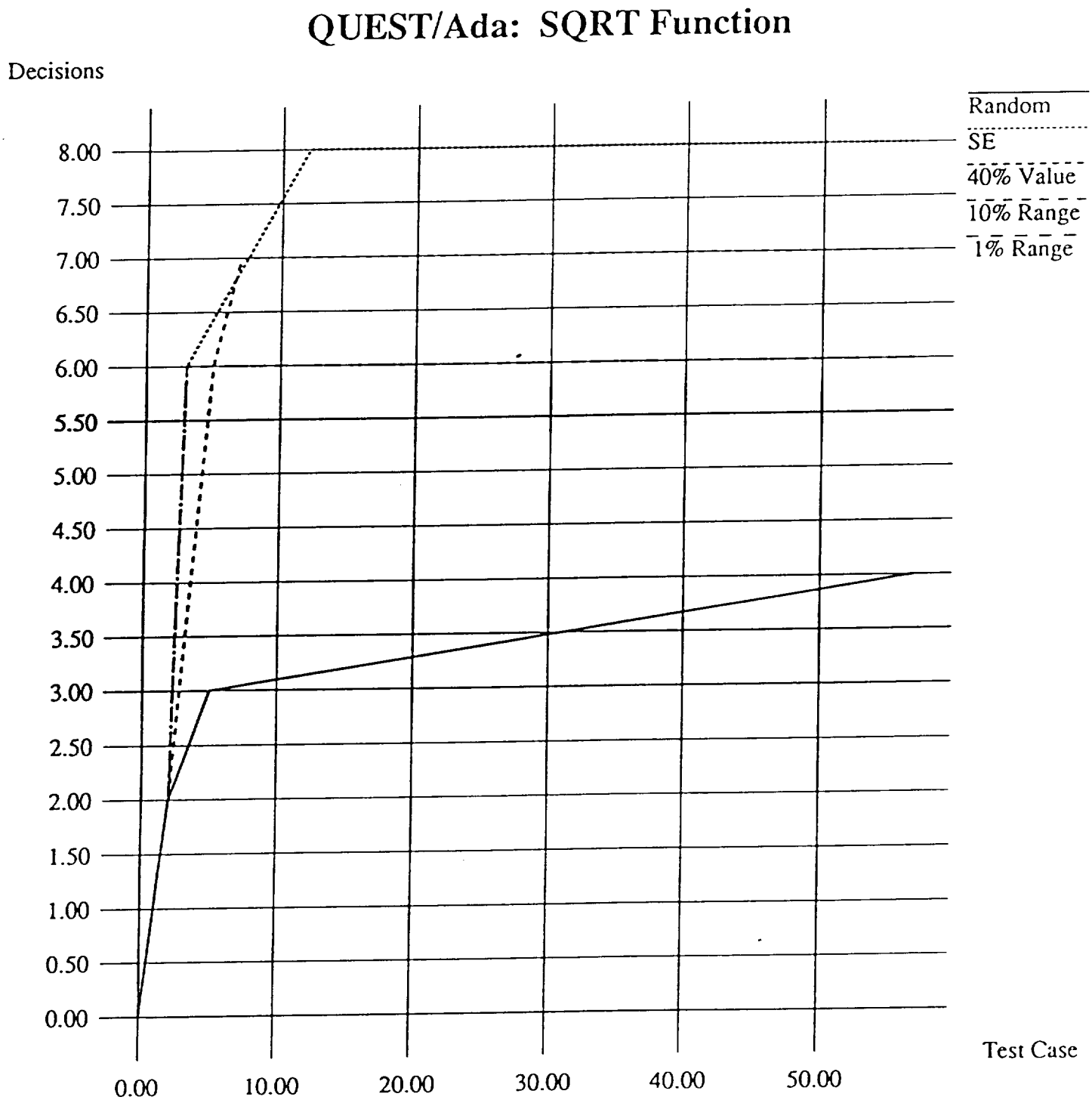

Figure 3.2f Rule Comparison for SQRT Function

Table 3.2 NASA Module Testing Results

Arc Tangent function:

Random		SE		40% Value		10% Range		1 %Range	
0	0	0	0	0	0	0	0	0	0
1	2	1	1	1	1	1	1		1
2	3	2	2	2	2	2	2		2
22	3	22	2	4	4	4	4		4
				5	5	5	5		22 6
				22	6	22	6		

Arc Sine function:

Random		SE		40% Value		10% Range		1% Range	
0	0	0	0	0	0	0	0	0	0
1	2	1	2	1	2	1	2	1	2
2	3	2	3	2	3	2	4	2	4
17	3	17	3	4	5	3	5	3	5
				5	6	4	7	4	7
				17	7	5	8	5	8
						17	8	17	8

Sine function:

Random		SE		40% Value		10% Range		1% Range	
0	0	0	0	0	0	0	0	0	0
1	4	1	4	1	4	1	4	1	4
2	5	9	4	2	5	2	5	2	5
4	8			4	7	3	6	4	7
6	9			5	8	4	7	5	8
9	10			9	8	6	8	9	8
						8	9		

Square Root function:

Random		SE		40% Value		10% Range		1% Range	
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
5	3	3	6	5	6	3	6	3	6
57	4	12	8	7	7	57	6	57	6
		57	8	57	7				

4.0 CONCURRENCY TESTING FOR ADA PROGRAMS

One of the main goals of concurrent program testing, debugging and analysis is to detect tasking errors -- mainly, deadlocks, communication errors, and concurrent access of shared variables.

There are two fundamental approaches to Ada concurrency testing: static analysis and task monitoring. In the static analysis, *all possible task states* of a program are explored and checked for tasking deadlocks [STR81, TAY83, DIL90]. A common problem of this analysis is the unmanageable number of states. In this type of analysis, a program is not actually run; it is simply analyzed syntactically. Because a syntactically possible state may not be semantically possible, a large portion of the analysis effort may be wasted. Another static analysis approach is to verify each task individually in local proofs and then check for task interference in a separate cooperation proof, all through symbolic execution [DIL90]. The local proofs determine the partial correctness of each task and identify communication interactions between tasks, while the cooperation proof verify mutual exclusion and the absence of deadlock.

In the task monitoring approach, a program is actually run [CHE87, GER84, HEL85]. A separate run-time monitor records the task states and interactions. Similar to the conventional software testing, "instrumentation code" must be inserted in the source program for the tasking information collection purpose. Unfortunately, the extra codes may result in an incorrect representation of the original tasking states and errors not detected [TAY88].

4.1 CONCURRENCY TESTING MEASUREMENT

One important aspect of software testing is the thoroughness of testing. However, because of the dynamic feature of a concurrent program the program testing coverage is difficult to measure. In particular, the task state space can be so large that it is impossible to compute its size. The literature review and research to this point has led to the following potential measurements for "concurrency" coverage:

1. Task entry coverage: Each syntactically identifiable task is recognized as a task unit. If a task contains other tasks, they are recognized as separate task units. This concept is analogous to statement coverage in conventional testing. The difference is that task units are identified instead of program statements. Most tasks must be called before they are activated. For this reason, task entry coverage measures *the completeness of tasks being called.* It is important to note that a task unit may be called by different program units. Therefore, complete task coverage does not guarantee complete statement coverage. This measure can also be viewed as rendezvous coverage.

2. Task calling statement coverage: A task calling statement requests service from a task unit. This measurement gives the coverage completeness of all possible communication links between tasks.

3. Task state space: If the size of possible (or feasible) state space can be computed, the coverage of state space may give a good measurement of the testing completeness.

4.2 DATA STRUCTURES FOR CONCURRENCY TESTING

Three kinds of information are needed for the proposed concurrency testing: program structure, active-task dependencies, and task coverage. The program structure presents the syntactical relationships among task units of the program under testing. The DIANA (Descriptive Intermediate Attributed Notation for Ada) package is used to provide this data structure in the proposed implementation. The active-task dependencies data structure records the dynamic behaviors of all active tasks. This information is used to analyze possible faulty behaviors, such as deadlocks and concurrent access of resources. A graph representation will be used for this purpose. The task coverage information indicates the completeness of testing. Task coverage is based on the testing goals. As described earlier, this may include the task entry, task calling statement, task space, or any combinations of these. Coverage tables will be used for this purpose.

There are two types of coverage tables in addition to the structure described above. The first type is the summary table which lists all task units. In this table, each task unit is marked either as covered or not covered, indicating the task entry coverage. The second type of table is for each individual task unit. It contains two columns. One column indicates the task units that the titled task may call, while the other column indicates the task units that may call the titled task. Illustrations for these tables are given in the QUEST Phase 2 report [Brown90].

4.3 TOOL REQUIREMENTS

In order to achieve and measure the task entry coverage and the task calling statement coverage, task dependency information of the tested program must be provided to the system. This information tells how a task can be activated. When a task is selected as a candidate for testing coverage, this task and its parent task must become active first. Although most tasks become active through task declaration, some run-time dependent tasks must be activated through program execution. This tasking dependency information also indicates the tasks that are called by each individual task. This is needed for the task calling statement coverage.

The DIANA package that is currently being used will provide all the needed information. One of the major purposes of DIANA is to provide an intermediate representation of an Ada program. A side benefit of using the DIANA package is that it may make automatic instrumentation possible within the QUEST project. This is because DIANA provides pointers from the structure nets to the source code. With the pointers, appropriate instrumentation statements can be inserted.

4.4 APPROACHES

The following tasks must be accomplished to achieve concurrency testing: (1) designing a coverage metric for the program under test, (2) developing a procedure for determining the next coverage candidates, and (3) performing tasking control and/or generating test data to drive the desired coverage.

The coverage measurement may use any of the criteria mentioned above. When the intermediate program representation is derived (e.g., by DIANA), a table-like coverage metric can also be built. This metric will be similar to the branch coverage table of the current QUEST/Ada.

The second task is to determine the next coverage candidates. A coverage candidate can be a single task unit or a particular sequence of task units. If the invoking sequence of task units is not specified, the sequence must be defined before the tasking behavior control or the test data can be determined. This task may be divided into two parts, coverage candidate identification and sequence (or path) identification.

The last phase is to perform the tasking behavior monitoring and control or to generate test data that will drive the desired coverage. These approaches will be described in the following subsections.

4.4.1 TEST DATA GENERATION APPROACH

Static analysis and task monitoring represent two extremes in Ada program testing or debugging. The static analysis approach attempts to explore and analyze all possible tasking states. On the other hand, task monitoring records and analyzes only one run of the program execution at a time. From another perspective, the static analysis approach analyzes the whole input space and the task monitoring approach analyzes only one point in the input space. Here, the input space represents any input parameters over which a user of the program has control. The search space for the static analysis is too large for reasonable effort, and the space for the task monitoring is only a point. A rational compromise is to settle somewhere between these two extremes.

Since each task monitoring cycle needs input data (or a test case) to drive it, a more thorough testing can be achieved by providing test data for more task monitoring cycles. If the test data is well designed, representative task states can be monitored and analyzed. While many task monitoring and deadlock analysis approaches have been reported, our research will emphasize test data generation for task monitoring since this shows the most promise for success consistently with the current QUEST/Ada approach.

The QUEST/Ada test data generator is designed for program unit testing. A program unit can be a task, subroutine, or program body. A set of test data is generated for a program unit to ensure branch coverage. During the execution of an Ada program, several tasks may be active at the same time, and these tasks may belong to different program units. For an active task that requires input data, it will be appropriate to use the test data generated for the involved program units. Since each program unit has a large number of test cases, combinations of these test data from various program units will provide a wide variety of "concurrency" coverage. The following section will demonstrate the methods to be applied to producing the required coverage.

4.4.2 IRON FISTED TESTING APPROACH

The fundamental philosophy of the proposed "iron fisted testing" is to drive Ada program execution in a way that the desired "concurrency" coverage can be achieved. When a task event happens, a specially designed scheduler will determine the sequence of tasks to follow. Possible actions include continuing the current task, blocking the current task, activating a blocked task, and forcing the execution to follow a particular direction. The decision is based on the current tasking state, the current coverage status, the program structure, the task priorities, and the desired goals. These criteria will be encoded in production rules as is currently done for test case generation. The preconditions of a rule define its applicability, and the consequences specify the actions to be taken when the rule fires. The following subsections explain the potential actions of the scheduler.

When multiple tasks are available for execution, task priorities will be used to determine which tasks should be executed. For this reason, priorities will dictate how the iron fisted testing proceeds under these circumstances. The ultimate purpose of these priorities is to achieve the desired goals efficiently. Sample priority assignment principles follow (in descending order):

1. A task which leads to the unblocking of other tasks,

2. A task which leads to a desired coverage,

3. A task which has not been executed before,

4. A task which may be called by other tasks, and the lowest priority,

5. A task which does not interact with other tasks.

From these principles it can be seen that various information are needed to determine a priority assignment. These include coverage tables, task states, and the program structure.

A task may be blocked naturally, due to the built-in Ada scheduler, or it may be blocked artificially by the iron fisted scheduler. A naturally blocked task must be unblocked by the built-in scheduler. An artificially blocked task must be reactivated by the iron fisted scheduler.

Before the iron fisted scheduler performs the tasking control, an initial execution of the program is required to provide data upon which the scheduler can function. This can be achieved by letting the program run freely for a limited time, e.g., 1 minute. During this time, the coverage information is recorded. After the time limit, the scheduler will perform the tasking control based on the achieved coverage to that point. Details of the design of potential scheduling policies are given in the Phase 2 report [Brown90].

5.0 TECHNOLOGY TRANSFER DOCUMENT

The purpose of the QUEST technology transfer document is to provide the software tools industry with the information needed to implement a production-quality QUEST system. Generally, this will require recording or otherwise adapting the prototype so that it can handle the full range of Ada constructs. While this is a very large task, it is not technologically difficult given the model which the current QUEST prototype provides.

This section is organized in order to facilitate a rapid understanding of QUEST by someone who has little experience with such tools. The first section presents the user interface such that immediate access to the system can be attained in order to enable the remaining parts of the document to be better understood. The second major subdivision presents a general description of the methodology employed by QUEST, i.e., a high-level view of the theory upon which QUEST is based. For more lower-level details, the reader is frequently referenced to the appropriate sections of other QUEST reports.

In the third major section of this document the QUEST system structure is presented. Input/Output Requirements Language [TBE84]
is used to define the system components and subcomponents. The interfaces (i.e., messages sent between component objects) are also described. This provides an indexed reference to the next section which describes the directories and the files which make up the QUEST system.

5.1 USER INTERFACE

The QUEST User Interface has been implemented in XWindows on networked Sun Workstations. XWindows allows the user to interact with the user interface through the use of a mouse and pulldown menus.

The initial QUEST window provides the user with a number of options. As shown in Figure 5.1, the main user interface contains options for four pulldown menus: Project, Testing, Reports, and Help. The three bars on this window indicate the progress of the testing (once testing is selected). Although the bars are given initial values at the start of the application, they may be changed by selecting an option from the Testing pulldown menu.

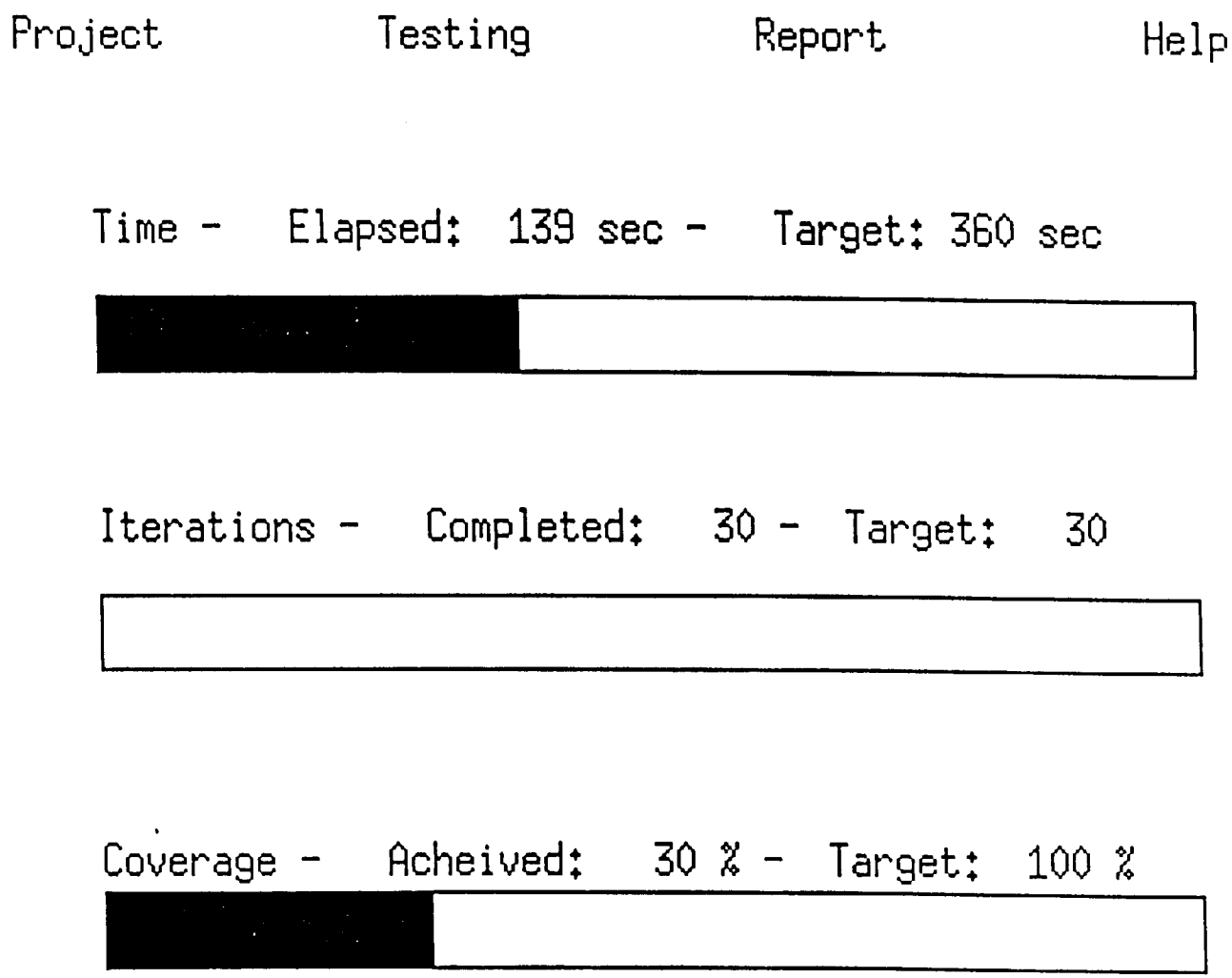

Figure 5.1 User Interface Screen

5.1.1 PROJECT SUBMENU

The Project Menu allows the "project" to be selected. A project is a grouping of one source module along with all of the supporting files needed for testing Ada. It must be created or selected in order to begin using the interface. Selecting a project will provide the user with a list of the Ada files in that project's directory. Once the user selects the file, it will be compiled and prepared for execution. The Project Menu also allows the user to create a new project. Other selections include closing projects, deleting projects, and exiting the user interface.

When the "Project" option is selected form the User Interface Screen the pull-down menu of Figure 5.1.1a will appear. These suboptions have the following functions:

New - creates an entirely new project.
Open - allows an existing project to be opened. This will produce the window shown in Figure 5.1.1b, which gives the user the ability to select the Ada module to be tested. Entries in reverse field are subdirectories. Their selection will lead to another similar window shown in Figure 5.1.1c.
Close - closes an open project.
Delete - deletes an entire project (not enabled).
Quit - restores control to the User Interface Screen.

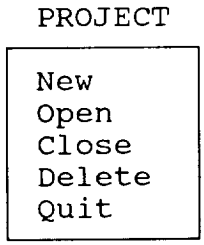

Figure 5.1.1a Project Submenu

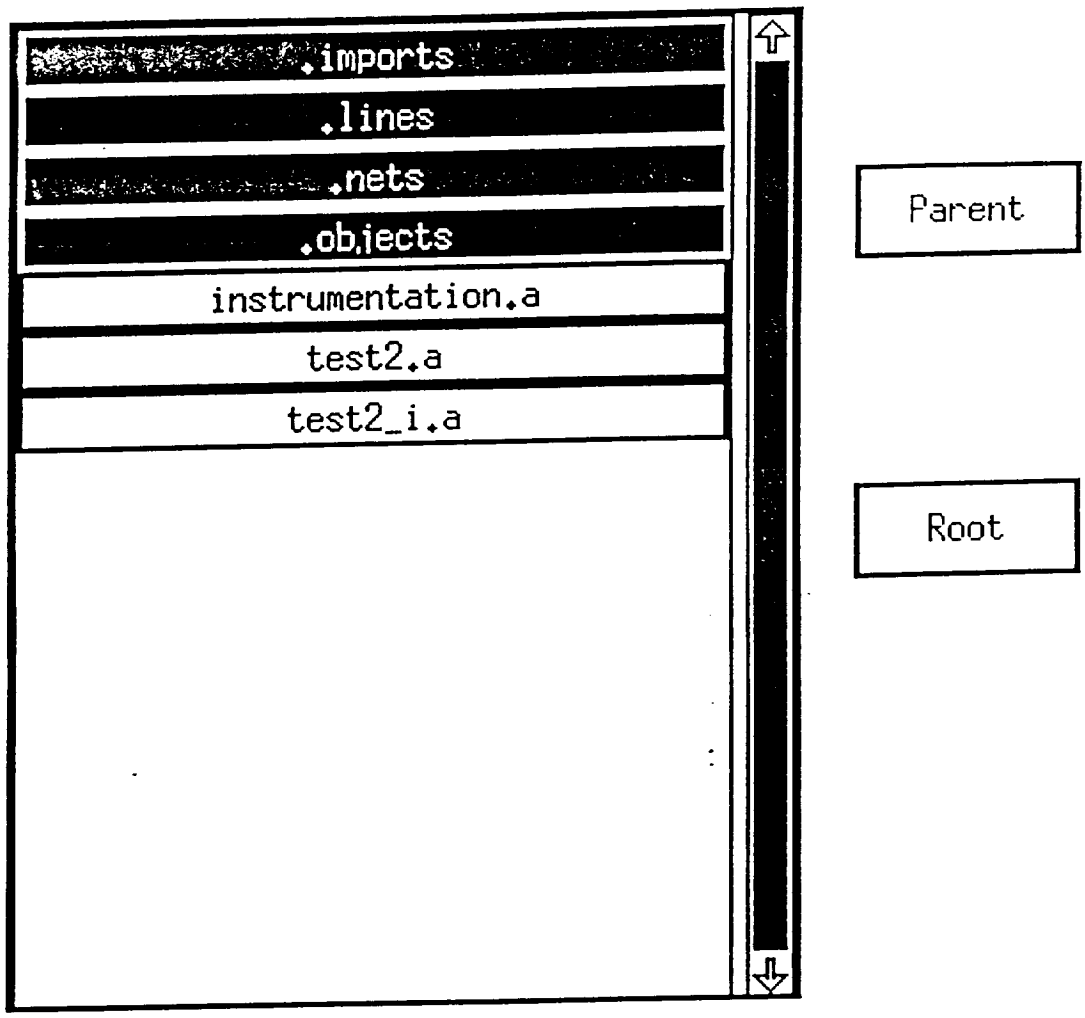

Figure 5.1.1b Directory Option Window

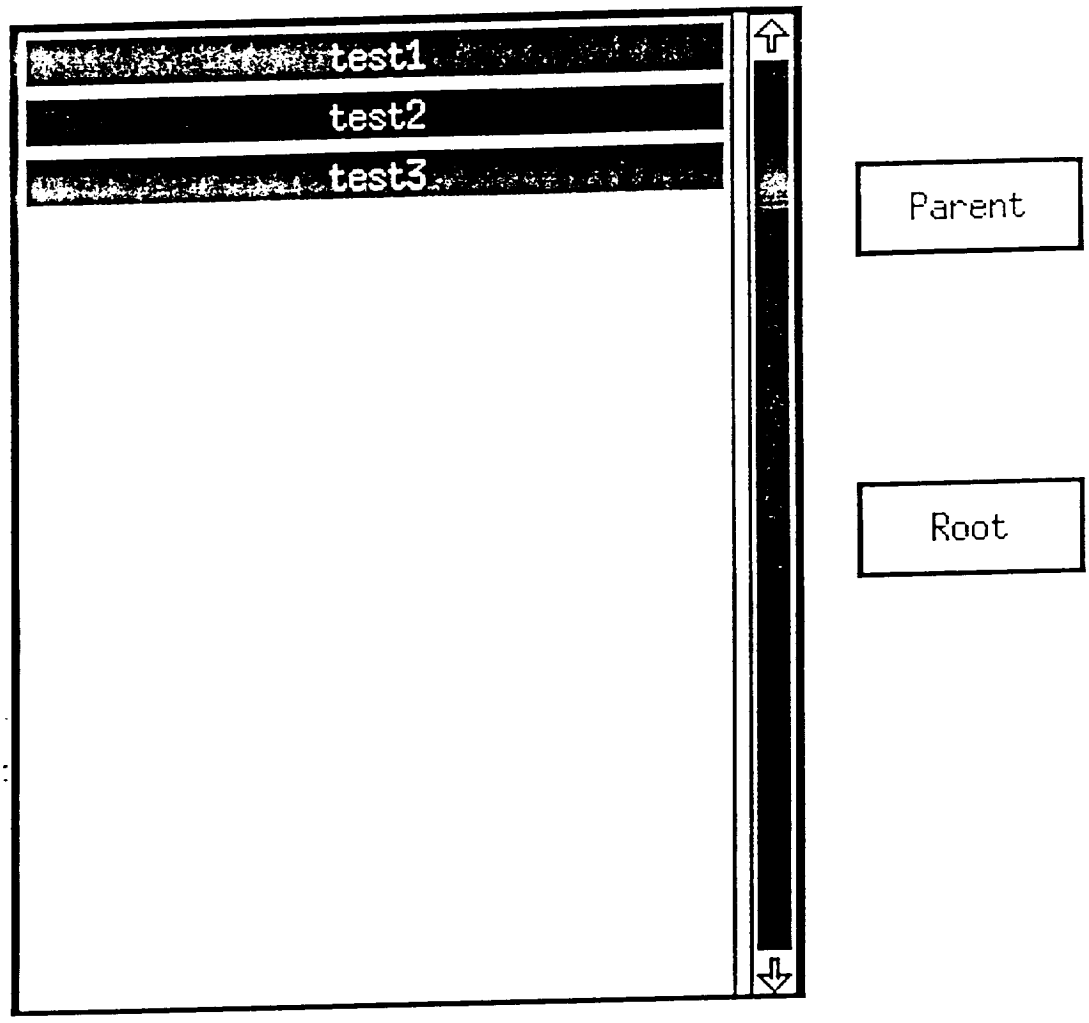

Figure 5.1.1c Subdirectory Option Window

5.1.2 TESTING SUBMENU

The Testing Section is the heart of the user interface. It allows the user to start testing, stop testing, and change the test metrics. When the "Testing" option is selected from the User Interface Screen, the testing submenu given in Figure 5.1.2a will appear.

To begin testing, select the "Start Testing" option given in the submenu. At this point, the instrumented code will be executed and the resulting data will be cataloged. After each iteration of compiling data, the bars on the User Interface Screen will be updated to reflect the progress of the test. Testing may be stopped anytime by selecting the Halt Testing option.

The two options on the Testing Submenu "Enter Test Case" and "Select Test Set" are not yet operational. The former option is the logical point at which the user can be prompted for the variable values of a user-defined test case. Similarly, the "Select Test Set" option would query the user for a file containing a number of test cases. The implementation of these options is essential to the finally functioning test system in that the user should have the flexibility to override the test case generator, especially for initial test case specification. However, while the implementation of these is quite labor intensive, it would contribute little of theoretical interest, and therefore it has not been included in the current prototype.

If the values for time, iterations, or coverage given on the User Interface Screen are not desired, they may be changed through the "Set Test Metrics" option. Once selected, the window will appear which is given in Figure 5.1.2b. Any of these three values can be altered directly on the screen.

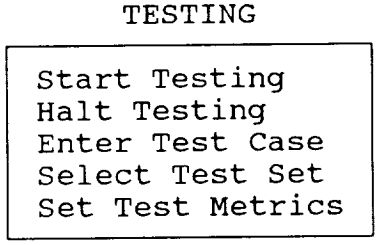

Figure 5.1.2a Testing Submenu

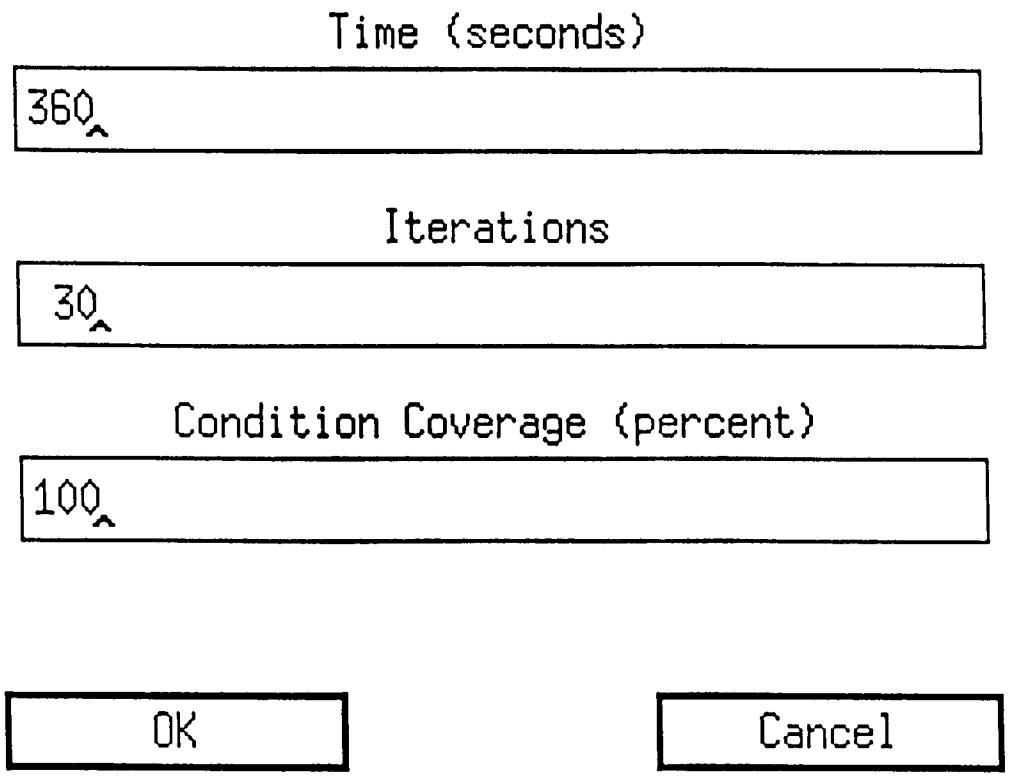

Figure 5.1.2b Set Test Metrics Window

5.1.3 REPORTS SUBMENU

The Reports Option is used to generate reports concerning testing which has already been done. When selected, the Reports Submenu given in Figure 5.1.3 will appear. Two types of report generators are available: Coverage Reports and Best Test Case Reports. Currently, these report generators are written for the VAX VMS file management system. Conversion to the workstation environment is expected in Phase 3.

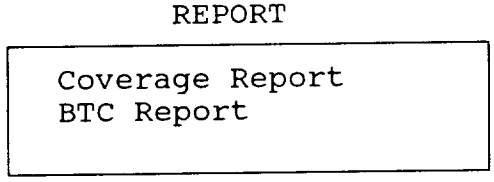

Figure 5.1.3 Reports Submenu

5.1.4 HELP SUBMENU

The Help Option is designed to provide information about the user interface. The user can select a general help or choose a keyword on which to find help. Since the help is a scrollable window, searches may be easily conducted for the information required.

5.2 METHODOLOGY OF THE QUEST/ADA PROTOTYPE

The QUEST/Ada prototype consists of five parts, which are discussed briefly below. Each will be described in greater detail in the subsections which follow the overview of the prototype.

5.2.1 QUEST/ADA PROTOTYPE OVERVIEW

The first step in testing a module of source code is to pass a file containing the source to the Parser/Scanner Module (PSM). The PSM is responsible for collecting basic data about the program, such as the names, types, and bounds of all of the variables, as well as the number of conditions and decisions found in the module. Additionally, the PSM is responsible for "instrumenting" the source code, which involves replacing each Boolean condition in the program with a function call to the Boolean function "RELOP" (see example instrumented code will be given below). Instrumentation also involves surrounding the test module with a "driver" or "harness". This harness is responsible for passing the test data generated by the rule base to the module under test, either as parameters or global information.

Once the source module has been scanned and instrumented, initial test data are prepared for it by the Test Data Generator (TDG). The TDG is an expert system designed to select the test data that will be most likely to drive a specific control path in the program. Four types of rules were considered and evaluated in the test data generator: random, initial, parse-level, and symbolic evaluation. Random rules, as the name implies, simply generate random test data. The generation of random data provides base data for the more sophisticated rule types to manipulate. Similarly, the initial rules generate simple base data from the information supplied from the parse. Parse-level rules, which are more sophisticated, rely upon the coverage table and best-test-case list developed by the Test Coverage Analyzer (see below). Parse-level rules implement the path prefix testing strategy described by Prather and Myers [PRA87]. Finally, symbolic evaluation rules extend this concept by representing each section of the program as an abstract function.

The symbolic evaluation rules utilize the coverage table and the symbolic boundary information. The work of the symbolic evaluator (SE) is divided into two parts -- developing and evaluating symbolic expressions. Using descriptions of the conditions in the module under test provided by the PSM, the SE develops symbolic boundary expressions in which each of the variables in a condition is represented in terms of the other variables. This boundary expression implicitly describes the point at which the input variables will cause the Boolean condition to evaluate to equivalence. Thus, by adding or subtracting a small value, epsilon, to the boundary, the inequality can be forced into each of its three states. After developing the symbolic boundary equations, the SE evaluates them using the test data as it appears at the time the condition is executed.

As mentioned above, the more sophisticated rule types rely on the Test Coverage Analyzer (TCA). The TCA provides two major functions: maintaining the coverage table, and determining the "best" test case for every decision. The coverage table maintains a list of each decision and condition in the module under test. Each decision and condition may have one of four mutually exclusive coverage states: not covered, covered true only, covered false only, and fully covered. This information is used by the parse-level and symbolic evaluation rules to determine which decisions or conditions need to be covered to provide complete decision/condition coverage. The best test case for each decision is determined by a mathematical formula describing the closeness of a given test case to the boundary of a specific condition. The test data generator rule bases modify the best test case to attempt to create new coverage in the module under test.

Finally, a data management facility exists within the prototype to simplify the user interface and report generation functions. This facility, known as the Librarian, is designed to be portable so that a user interface can be developed on several machines by accessing the librarian in a similar fashion. Additionally, the Librarian acts as a data archive so that regression and mutation testing may be implemented using previously generated test cases.

These functions act together to provide a prototype environment for the rule-based testing paradigm. Each one of the major parts of the prototype is described in greater detail in the following sections.

5.2.2 TEST DATA GENERATOR

As designed, the QUEST/Ada system's performance is determined by two factors: (1) the initial test case rules chosen to generate new test cases, and (2) the method used to select a best test case when there are several which are known to drive a path to a specific condition. If the user does not supply an initial set of test cases, then they are generated by rules that require knowledge of the type and range of the input variables. These initial test cases are generated for these variables to represent their upper and lower values as well as their mid-range values, i.e., (upper limit - lower limit)/2.

5.2.2.1 BEST TEST CASES

The objective of the Test Data Generation (TDG) component of QUEST is to achieve maximal branch coverage. In order to assure the direction of test case generation to be fruitful, a branch coverage analysis is needed. The coverage analysis of this framework follows the Path Prefix Strategy of Prather and Myers [PRA87]. In this strategy, the software code is represented as a simplified flow chart. The branch coverage status of the code is recorded in a coverage table. When a branch is driven (or covered) by any test case, the corresponding entry in the table is marked with an "X". For detailed examples see Section 3.2 of the Phase 2 report [Brown90].

Problems arise when there is more than one test case driving the same path. If several cases are used as the basis for subsequent test case generation, efforts are likely to be duplicated, which is not efficient. Since an automatic case generator can generate a large amount of cases, it would be necessary to quantify the "goodness" of each case and use the "best" case as the model for modification.

The objective of modifying the best test case is to generate a new case which will cover the uncovered branch of the targeted condition. For this reason, the selection of a best test case will directly affect the success of test case generation.

Consider the typical format of an IF statement: IF exp THEN do-1 ELSE do-2. The evaluated Boolean value of exp determines the branching. Exp can be expressed in the form of: lhs <op> rhs. Lhs and rhs are both arithmetic expressions and <op> is one of the logic operators such as <, >, <=, >=, <>, and =. The goodness of a test case, t1, can be measured in terms of the closeness of lhs and rhs, provided that t1 is used as the input. When this measure is small, it is generally true that a slight modification of t1 may change the truth value of exp, thus covering the other branch. With this definition, a test

case that yields the smallest measurement is considered to be the best test case of the condition under consideration.

This closeness measurement has a serious risk, however. Recall that a set of new test cases is generated based on the best test case of a partially covered condition (called target condition), and the intent of the new test case set is to cover the uncovered branch of the target condition. Although we define the slightness of modification of a test case as its goodness, this measure is computed based on the target condition only. A slight modification to the lhs and rhs of the target condition may not have the same meaning to those conditions on the path. This may result in unanticipated branchings along the path, therefore losing the original purpose of the new cases. In order to reduce the likelihood of unanticipated branching, a test case's goodness measure should also consider those conditions that are on the path. An example further explaining this requirement is given in Section 3.2 of the Phase 2 report [Brown90].

Although the measurement that considers the complete path seems more appropriate than the one that considers the target condition only, this is impossible to prove theoretically, since both definitions are derived heuristically.

When a test case is run in the test case analyzer and it reaches a condition that is either partially covered or not covered at all, its goodness value is computed. This value is then compared with the goodness value of the current best case, if there is one. If its value is smaller, this test case replaces the original case and becomes the new best case. In the QUEST prototype the test case analyzer keeps more than one test case for each partially covered condition. That is, the second, the third, and the fourth best cases are also kept. This provides alternatives for the test case generator when the original model does not yield new coverage. At this point the test data generator procedures will be described in a general way. Detailed descriptions are contained in Section 3.2 of the Phase 2 report [Brown90].

5.2.2.2 TEST DATA GENERATOR PROCEDURE

When a new test case is generated, it is with the intention of covering a particular branch. Based on the best test case of a targeted partially covered condition, a slight modification to the case is made with the intent to execute the uncovered branch of the target condition. The main issue in the research has been the establishment of methods for efficiently performing this modification.

The following subsections discuss some heuristics that can be used for modification to generate new cases, including: (1) fixed percentage, (2) random, (3) condition constants and (4) symbolic evaluation.

5.2.2.2.1 FIXED PERCENTAGE MODIFICATION

One way of generating new cases is to modify each parameter of the best test case with a fixed percentage of each parameter's ranges. The percentage can be any one of or any combination of 1%, 3%, 5%, 10%, etc. Several different combinations can be used at the same time. This would provide more new cases. After a new case is generated, it must be checked to ensure that each variable is within its range.

5.2.2.2.2 RANDOM MODIFICATION

This method modifies the best test case in a random way, i.e., the modification percentage is random. Each new case must be checked for its validity before it is stored. Random modification can be done in several ways. That is, in each new case, one or several variables can be modified. Combinations of these modifications provide more cases and may cover more branches.

5.2.2.2.3 MODIFICATION BASED ON CONDITION CONSTANTS

This method generates new cases based on the constants appearing in a condition. Depending on the number of constants in a condition, different rules can be applied. For example, if there is one constant and one input variable in a condition, then generate a new case by putting the constant in the position of the input variable in the best test case. This rule is designed for conditions of the form: x <op> C, where C is a constant. Similarly, for two constant conditions, e.g., $x+C_1$ <op> C_2, three new cases can be generated. They are C_1+C_2, C_1-C_2, and C_2-C_1. Implementation of this kind of heuristic has been reported in a separate paper [DEA89].

5.2.2.2.4 MODIFICATION BASED ON SYMBOLIC EVALUATION

Another approach to new test case generation is to determine the boundary that separates the true and the false values of a condition, say D. Effort is then directed to modify the best case to cover both sides of the boundary. Since the evaluation of D can only be externally controlled by input parameters, say x, y, and z, a meaningful way of expressing the boundary would be defining it in terms of x, y, and z.

Remember that new test case generation should be based on the best case (x_1, y_1, z_1) and the modification should be as small as possible. A simple strategy would be to modify only one variable at a time. For example we can modify x and keep y and z unchanged. In this case, the condition boundary expressed for x should be used, i.e., $x_b = f1(y,z,v_1,v_2, ...)$. In order to compute the desired value of x at D, use the actual values of y, z, v_1, v_2, ... just before D is evaluated. The computation provides the desired boundary value of x at

condition D. Three new cases can be generated to cover both true and false branches: (x_b, y_1, z_1), (x_b+e, y_1, z_1), (x_b-e, y_1, z_1). Here, e is a small positive number, e.g., e = (range of x) / 100. Similarly, this case generation procedure can be applied to variables y and z.

In this procedure, an undesirable assumption is made. It is assumed that x (or y or z) would not be modified between the entry point and the target condition D. This may not be valid at all. If an input variable value is modified by the program before reaching the target condition, the precise computation of the boundary may lose its purpose. Whether an input variable has been modified or not can be checked easily. For example, if (x_1, y_1, z_1) is a test case of the procedure and (x_c, y_c, z_c) are the actual values of x, y, and z just before condition D is executed, input variable modification can be checked by comparing these two sets of values. If a variable, e.g., x, has not been modified, i.e., x_1 = x_c, then the computed condition boundary, x_b, can be used directly for new case generation.

Now, the question becomes: what can be done if an input variable has been modified? If the desired boundary value of x at condition D is x_b, this value must be inverted back through the path that leads to condition D. Through this inversion, the value of x at the entry point can be found. However, this involves a complex path predicate problem which does not have a general solution [PRA87]. Heuristic approaches toward solving this problem will be presented below.

Consider the following situation. The input value of x is x_i, the actual value of x just before condition D is x_c, and x_i < > x_c. This means variable x has been modified before reaching D. Assume the condition boundary of x at D is x_b. In this case, we might surmise that input x should be changed from x_i to an unknown value x_u such that, just before reaching D, x will be changed from x_c to x_b. Since we do not know how x is modified along the path, precise modification to x at the entry point cannot be computed. However, an approximation can be derived. At condition D, the desired value of x is x_b and the provided value is x_c. We may consider x_i is off the target, i.e., the condition boundary at D, by the following percentage of $|x_b - x_c| / (2*MAX(|x_b|, |x_c|)) * 100$ %. Following this measurement, we can modify input x with the same percentage.

Another way of approximating the input boundary value is to assume a linear relationship between x_c and x_i. In this situation, the approximated boundary value for x at the entry point would be x_b*x_i/x_c. Three new cases can be generated for being on or slightly off the boundary.

The order of execution or control flow of the Symbolic Evaluator to generate boundary-values facts follows. The Symbolic Evaluator initializes a value for each variable from the Parser/Scanner to NIL, evaluates each conditional expression, generates a boundary condition, evaluates each boundary condition with conditional values (from the Intermediate Results file), and replaces the NIL value with the actual boundary value.

The input and output facts of the Symbolic Evaluator are contained in a series of lists. The list of variables from the Parser/Scanner are created as a fact in "names X1 X2 ... Xn". The Intermediate Results file is used to create conditional values stored as "val-at-cond Y1 Y2 ... Yn" facts. The "val-at-cond's" are the values at the decision and condition point for this evaluation. The Parser/Scanner generates the conditional expressions in infix notation for conversion to "cond-expr Z1 Z2 ... Zn" facts.

During execution, the Symbolic Evaluator sets a value for each variable to NIL (list-of-nils). The boundary expressions are then generated and evaluated. New values replace the NIL value if they are found; they are placed in the "boundary-values" listing. The boundary values are submitted to the expert system for further evaluation if this is required.

When using the symbolic evaluation rules, the Test Data Generator requires the intermediate results from the execution of the instrumented code and the conditional expressions from the Parser/Scanner in order to generate facts and then execute. The intermediate results and conditional expressions are put into files for the Test Data Generator to read so that it can generate the required facts. The files are read, facts generated, boundary results created, and new test cases generated. The files are then closed awaiting new intermediate results.

In this section, several heuristic rules have been presented. It is likely that each rule is effective in certain situations. If several rules are applied to a program, they will complement each other and yield better coverage. The specific facts, given in CLIPS code, as well as the CLIPS salience levels are given in Section 3.2 of the Phase 2 report [Brown90], along with additional details on control flow and the symbolic evaluator and the symbolic evaluator interface.

5.2.3 PARSER/SCANNER

Note that with the exception of the DIANA interface (Section 5.2.3.5), the instrumentation function are not part of the QUEST/Ada prototype. See Section 5.2.3.4 for more details.

5.2.3.1 BASIC INSTRUMENTATION

Whereas static information concerning the Module Under Test (MUT) is provided to the Test Data Generator via the Parser/Scanner Module, run-time information is obtained through the use of function calls inserted into the original source code. These function calls are placed at the various decisions throughout a program in order to determine the set of paths executed by a particular set of test data. The information acquired by the function calls is written to an intermediate file that is read by the Test

Coverage Analyzer and converted to forms that are usable by the Test Data Generator and the Librarian.

The decisions that are instrumented by QUEST are those consisting of Boolean expressions in the following form:

LHS <relational operator> RHS.

These expressions are replaced by function calls that evaluate their truth value and return this value to the calling program.

A line of information is written to the intermediate file indicating the test number, the decision and condition number, the truth value of the expression, and the values of the left hand side and right hand side of the expression. These functions have the following specification:

```
function relop(TestNum:integer;
        DecNum:     integer;
        CondNum:    integer;
        LHS:        Expr_type;
        OP:         Relop_type;
        RHS:        Expr_type) return BOOLEAN;
```

The functions are encapsulated in Ada GENERIC packages to facilitate parameter passing and input/output of user-defined types. Currently, packages are available for integer, enumerated, floating point, and fixed point data types.

The MUT is surrounded by a harness (i.e., driver program) that controls its execution during testing. The driver is responsible for reading the test cases from a file and passing this data to the MUT as arguments. Also, global data, out parameters, and return values are written to a file for user inspection and regression test purposes.

5.2.3.2 INSTRUMENTATION FOR SYMBOLIC EVALUATION

Instrumentation for symbolic evaluation requires that the intermediate values of the input parameters to the MUT be obtained at each decision in the program. Since Ada is a strongly typed language, it is not possible to simply pass these parameters to the instrumentation package because the number and types of the parameters vary according to the makeup of the MUT. Also, it is not possible to declare the procedure as SEPARATE to the instrumentation package, since the procedure must be declared inside the MUT in order for the parameters to be visible. This problem was circumvented by creating a procedure within the module under test and passing the procedure as a GENERIC to the

instrumentation package. The procedure only needs a single parameter -- the name of the file to which the output is to be directed.

5.2.3.3 INSTRUMENTATION FOR MULTIPLE CONDITIONS

Instrumentation for multiple conditions requires the instrumentation package to be extended to include a function to determine the overall truth value of a decision. For example, the following decision:

IF (a < b AND c > d) THEN

would be translated to the following statement:

```
IF decision(TEST_NUM,and(relop(TEST_NUM,1,a,LT,b),
                         relop(TEST_NUM,2,c,GT,d))) THEN
```

The function relop() acquires information about the individual conditions, while the function decision() acquires information about the overall decision.

5.2.3.4 AUTOMATIC INSTRUMENTATION

The instrumentation described here is currently being performed manually. Although automatic instrumentation could be performed during the execution of the Parser/Scanner Module, its implementation would have required considerable effort which would have greatly hindered progress on the other substantial areas of the research and prototyping. Given the instrumentation specified here, the development of an automatic instrumenter is seen to be a relatively straightforward task for those in the industry who are specializing in the design and development of Ada compilers. In fact, this could be integrated into the compiler and debugger tools in a very efficient manner. For these reasons, it was decided that prototyping of the automatic instrumentation would not be pursued immediately. However, the requirements for automatic instrumentation are clear in the manual examples which were employed to test the QUEST system. Examples of instrumented programs and source code for the instrumentation packages may be found in Appendix B of the Phase 2 report [Brown90].

5.2.3.5 DIANA INTERFACE

The Parser/Scanner Module has four primary functions:

1) Compile a List of executables,
2) Extract input facts,
3) Extract condition facts, and
4) Instrument the source module.

The Verdix Diana interface is used to retrieve this information from an Ada library. Diana is an abstract data structure containing information about an Ada source, and a set of procedures for getting information from the structure. Once the user has selected the system to be tested, the system is compiled using the -F (full Diana) option of the Verdix Ada compiler. This option ensures that no information about the source gets "pruned" out of the Ada library.

After the system is selected, the user is presented with a list of all of the executable modules in the system. This list is created by the Parser/Scanner Module. The PSM searches the Diana net, builds a linked list of the executables, and then passes it to the interface to be presented to the user.

Once the module to be tested has been selected, the PSM traverses the Diana net for the module and retrieves information about the input to the module. This includes facts about parameters and about global data used in the module. These facts are saved to a file to be read later by the Test Data Generator. The saved facts include:

Parameter Name	Type	Low Bounds	High Bounds

They are formatted as assertions to CLIPS [CLI87]:

```
( parser_scanner_assertions "<modulename>"
   ( names <parm1_name> <parm2_name> ... <parmn_name> )
   ( types <parm1_type> <parm2_type> ... <parmn_type> )
   ( low_bounds <parm1_low> <parm2_low> ... <parmn_low> )
   ( high_bounds <parm1_high> <parm2_high> ... <parmn_high>)
)
```

Facts about every decision in the module are gathered and written to a file. The Diana net is traversed in search of every decision in the module. Each decision is given unique number (dec#) as is each condition within a relation (cond#). These facts are formatted and saved for input into the Test Data Generator as follows:

```
( decision_condition_facts "<module_name>"
   <dec#> <cond#> ( <formatted condition> )
   <dec#> <cond#> ( <formatted condition> )
      ...
)
```

Instrumentation at each condition in the module must provide information about the results of the condition test. Currently, this instrumentation is done by hand. While automating this process is a fairly straightforward, it would require more manpower than is available on this project. The format of the instrumentation is expected to change as new requirements are received. The current format of the instrumentation function is:

relop (<test#>, <dec#>, <cond#>, <LHS>, <op>, <RHS>)

The relative operation function 'relop' takes as parameters the test number, decision number, condition number, left-hand-side of the condition, right-hand-side of the condition, and the operation to perform. It writes to a file called "INTERMEDIATE.RESULTS", which is later read by the Test Coverage Analyzer. The data written includes the test, decision and condition numbers, the left and right sides, the result of the operation (TRUE or FALSE), and the test data which caused this condition to be evaluated. It is encapsulated in Ada GENERIC packages to facilitate parameter passing an input/output of user-defined types.

'Relop' returns the results of the condition evaluation, so that it can be inserted as a function call in place of the condition. For example,

if (y*10<3) then ...

would be converted to:

if (relop (1, 5, y*10, 3, "<")) then ...

5.2.4 COVERAGE ANALYZER

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 first category of rule reflects only type (integer, float, etc.) information about the variables contained in the conditions, since they generate new test cases by randomly generating values. As implemented, these rules determine lower bounds, upper bounds, and types of the variables. A random value of the same type is generated, and the value is checked to be sure it is within the range for the variable.

The second category of rule attempts to incorporate information from three sources: (1) that which is routinely obtained by a parse of the expression that makes up a condition (such as variable types and ranges), (2) information about coverage so far obtained, and (3) best test cases from previous tests. A typical rule for this category would first determine bound and type information associated with a variable, calculate this range, and then generate new test cases incrementing or decrementing the variable by a small fraction of its range, and checking to see that the result is still in bounds.

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, a value is sought that will not alter the execution path to the condition. In addition to best test case selection, this rule base has additional knowledge to generate new test cases. The values of variables at a condition are compared with input values of the variables used to reach that condition. This added information is incorporated in the generation of new test cases.

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 is x_i. We do not know how x is modified along the path leading to the condition since the value of x on input may differ from the value of x at the condition. However, we are able to establish that the value of x at the condition is x_c. Provided the values lie in the limits allowed for values of x, the new test case is chosen as:

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

where e is either 0 or takes on a small value (positive or negative).

In general, these rules first match type and symbolic knowledge about the condition, information from the coverage table, and information about the values of the variables at the condition. Using this information the value required to alter the condition's truth value is symbolically computed. The new test case is generated by the formula given above, which supposes that a corresponding linear change will occur in the value of x from its initial value. The value of x is altered slightly in order to attempt to cross the boundary but not change the execution path to the condition.

5.2.4.1 AUTOTEST AND THE TEST COVERAGE ANALYZER

The purpose of the Autotest module is to coordinate the activities of the Test Data Generator (TDG), the module under test (MUT), and the Test Coverage Analyzer (TCA). Autotest repeatedly calls the above procedures until all of the required test packets are complete. The primary job of TCA is to supply the TDG with the best test cases which have been used to execute the MUT. It also accumulates data for reports after the test and archives the results. A best test case is chosen for each condition in the MUT. There can be several different methods for choosing the best test case. Currently, two methods have been implemented. The first is to calculate the distance each test case is from a border of the condition in order to select the case which is closest to the border. The second method involves the above procedure augmented by steps for the avoidance of previously encountered conditions. In this approach test cases are selected for closeness to the current

condition and distance from all of the previous conditions. The methods for selecting the best test cases are more fully described below.

The TCA keeps a coverage table entry for each condition encountered in the MUT. If a condition has not been encountered before, a new entry is created in the table. If it has been encountered before, but with a different Boolean result, it is updated to indicate complete coverage. The coverage statistics are based on the number of conditions in the module under test, the number that are partially covered, and the number that are completely covered.

Each condition entry in the coverage table contains references to the best test cases for that condition. When a condition is first encountered, the driving test case is the only test case for that condition; thus it is the best. As long as the condition is only partially covered, the TCG will attempt to generate test cases which continue to exercise the condition. When this occurs, the current test case will replace the previous best test case if the criteria being applied indicate that it is "better." The table is not altered for completely covered conditions since the TCG considers them to be completed.

After all of the test cases for a particular packet have been viewed and used to update the coverage table, the table is searched for partially covered conditions, and the associated best test cases are returned to the test data generator.

Test case generation rule groups may be exhausted before a new coverage is achieved. This failure can be attributed to two factors: inappropriate modification, and inappropriate best test case. This former factor may be solved by adding more rule groups. The second factor must be solved by selecting an alternative test case.

Since the selection of a best test case is based on heuristics, it may not be appropriate for some situations. For this reason, instead of keeping the best test case only, several "good" test cases should also be recorded for a partially covered condition. These cases can be ranked according to a goodness definition or selected from different goodness definitions. When a best test case has exhausted all case generation rules and no new coverage is achieved at the target condition, an alternative case will be used.

A branching decision may contain two or more Boolean conditions. This kind of decision is called a compound decision. It can be simplified into a form of IF A AND/OR B THEN do-1 ELSE do-2. A and B are both Boolean conditions and can be in a compound or simple form. A compound form contains at least one AND/OR operator. A simple form can be either a Boolean variable or an arithmetic expressions with a comparison operator, e.g., <, >, =, etc. Like a simple decision, two things must be considered for the compound decision: goodness measure of a test case at a decision, and test case generation rules. These will be considered in the following two paragraphs.

If a condition contains Boolean variable(s) only, the test case goodness measure should be based on the sum of condition boundary closeness along the path leading to the target condition. Since only Boolean variables are involved, closeness measurement cannot be done at the target condition. However, if there is at least one arithmetic expression in the condition, a normalized boundary closeness measure can be used. For example, consider a test case, (x=12, y=-8, and z=8), and a statement, IF (x > = 10) OR (y =< -10) THEN do-1 ELSE do-2. The boundary closeness measure of each individual term is calculated first. For the first term, (x > = 10), the measure is |12 - 10| / (2 * MAX(|12|, |10|) = 2/24; for the second term, (y =< -10), the measure is 2/20. The normalized measure is simply the average of these two measures. At this point earlier definitions of goodness can be applied.

In a decision containing multiple conditions, the negation of the Boolean conditions is not trivial. Consider the following two situations.

(1) IF a_1 THEN do-1 ELSE do-2

(2) IF a_1 and/or a_2 and/or a_3 THEN do-1 ELSE do-2

In (1), a change of the branching can be achieved simply by changing the Boolean value of a_1. On the other hand, in (2) the branching cannot always be modified by changing one item. Since there are three conditions in (2), there are eight possible combinations of the Boolean conditions. Among these combinations, some lead to do-1 and some lead to do-2, depending on the context of the problem. When a branch is targeted for further coverage, it will be required to assign Boolean values to all of the terms, i.e., a_1, a_2, and a_3. This assignment is not as simple as looking up the truth table of the condition. Since we try to minimize the modification of a best test case, this must also be considered in the truth value assignment of each condition.

Once the assignment to each condition is determined, test cases must be generated to satisfy the requirement of each condition. Unfortunately this may involve solving a set of predicates which has been recognized as an extremely hard problem, as referenced above. In order to simplify the test case generation, a set of heuristic rules are applied (see the Phase 2 report [Brown90] for details).

5.2.5 LIBRARIAN

The librarian routines for the Quest/Ada environment provide methods to easily archive and restore data for a particular test set. The librarian is implemented in three parts. The first is the code specific to manipulation of indexed records. This code has been isolated as much as possible to allow it to be changed if necessary. Currently it uses a set of shareware B-tree routines known as BPLUS to manage indexed files. The second part of the librarian code is the collection of librarian primitives. These primitives serve as an

abstracted interface to the specific file manipulation routines. This makes it easier to replace the code for managing indexing while keeping the same coding style for calling the librarian. The third and last part of the librarian is the code written specifically to manipulate QUEST/Ada files. The first two parts are mostly free of application-specific code, allowing them to be reused for other projects. In discussing the librarian and its design, the QUEST/Ada implementation will be used as the main example.

This section will continue by presenting some basic concepts employed by the librarian component of QUEST. A second section will detail the use of the Librarian. Some intricacies of these routines will then be described, after which appears some notes on its portability. The librarian routines are given and described in Appendix C of the Phase 2 report [Brown90].

5.2.5.1 BASIC CONCEPTS

A collection of binary data files contain records which represent information that has been archived from QUEST. These data files are known as "flat files" because they do not contain indexing information. Separate files exist to aid in indexing the data files. The name of an indexing file is the name of the data file concatenated by a "key number" assigned by the librarian which indicates the index file represented. Key numbers start at zero (which is usually the unique key for the data file). For example, if the file name was test1.dat, the index file name for key number zero would be test1.dat00, and the index file name for key number one would be test1.dat01.

All of the files are collected under the same directory. For QUEST/Ada, the file names are constructed by beginning with a given system name and concatenating onto it an extension representing the data contained in the flat file. For example, if the system name was XXX, the file names would be:

Coverage Table:	XXX.COV
Intermediate Results:	XXX.MED
Test Data:	XXX.DAT
Test Total Results:	XXX.RES

Again, the index files for the data files are the same except that the key number is tacked onto the end of the file name.

All of the librarian routines return a result code. Generally, if the return code is below zero, an error has occurred. If the return code is zero, the function has executed without any bothersome events. If the return code is greater than zero, some event has occurred which might be important information for QUEST users (an end of file, for example). All of the return codes are defined in the header file librarian.h by #define statements.

A data file can have more than one key. This simply means that the data file has an additional index file that can be used in another way to search through the data file. An index file can contain either unique or non-unique keys. At least one index file (usually number 00) should be unique so that specific records can be found. The keys are a composite collection of members in the data record.

5.2.5.2 USING THE LIBRARIAN

Prior to use, the librarian must be initialized, and the function lib_init() is called to allow the librarian to organize its data structures. The routine lib_directory() may be called to set the directory path in which the librarian files should be put. The function lib_set() is then called to establish which archive is to be opened or created. To start an archive from scratch, it is a good idea to call lib_remove() after calling lib_set() so that all existing archive files can be deleted.

After an archive has been set, its data files can be opened. The function lib_open() is passed a number representing the file to be opened. A number of options exist to read records from the file. Before attempting any read (including the initial sequential read), call the routine lib_set_key() to tell the librarian the index file by which the data file will be indexed. Sequential reading is enabled by using two steps. First, call lib_read() with the mode LIB_FIRST_REC to rewind the offset into the index file to the first record. This will also retrieve the first record from the file, if possible. To read all records after the first, call lib_read() with the mode LIB_NEXT_REC. This can be continued until the return code from lib_read() is LIB_EOF. To read keyed files, first call lib_set_key() to set up which key and which key components are to be employed for searching. Then call lib_read() with one of two modes: LIB_FIRST_MATCH or LIB_NEXT_MATCH:

LIB_FIRST_MATCH	will search the index file for the first occurrence of a matching key and if successful, it will retrieve the data record.
LIB_NEXT_MATCH	is used for index files in which the keys are not unique: more than one record can have the same key.

LIB_FIRST_MATCH is used to find the first match, and lib_read() can be called with the mode LIB_NEXT_MATCH to find all subsequent matching records. When no more records exist, LIB_NO_MATCH is returned.

Writing records to a file is much the same. First, all of the key contents for the record must be established by calling lib_set_key() for each one. This is essential. Upon calling lib_write(), all keys for the record are assumed correct and written out to their respective index files. This means that if a record has three keys, then lib_set_key() needs to be called for key 0, key 1, and key 2. Then the record can be saved via lib_write(). Note that lib_write() might "fail" if a particular key is supposed to be unique but already exists in the index file. In this case the data record is not written to the data file.

The function lib_close() should be called when record manipulation for a data set is complete. Under the BPLUS indexing system, it is **very** important that open files are closed. This is due to the indexing routines employing local "caching" of index information. If the files are not closed, this caching information may not be written out, and the index file can become inconsistent. The routines to terminate association with an archive or to shutdown the librarian determine if files are still open, and if so, they close them.

The function lib_open() is additive for a data set. If lib_open() is called more times than lib_close() is, a data set has a positive open count. It will not actually be closed until the same number of calls to lib_close() as there were to lib_open(). On shutdown, any files with non-zero open counts are considered opened, and an attempt will be made to close them.

5.2.5.3 DETAILS OF THE LIBRARIAN CODE

The librarian is designed to rely on another set of code to do the detailed work of creating indexes into a file. The librarian routines merely take a binary collection of data and save it somewhere, leaving a method to quickly find the data again later. The librarian is designed using the BPLUS collection of B-tree index file management routines.

Any given binary data record must possess the following attributes:

1. A data set number,
2. A set length (in bytes),
3. A set number of keys (at least one),
4. A data file to be stored in, and
5. Components that are used to create keys.

The librarian routines use the data set number for an index to access a global structure called lib_glbl. This global structure is very important because it is used to store descriptive attributes about each active file. This includes record size, number of keys, and the keys that have been set for the given record. Currently, lib_glbl is initialized in the function lib_b_setup(), which is called during execution of lib_set(). The keys for a record, although likely made up of components within the record, are not stored with the record in the data file. The function lib_set_key() needs to be called for each key in a record before the record is written out. Each time lib_set_key() is called, the associated key string in lib_glbl is updated.

The global lib_arch is used to keep track of less specific details, like the archive directory, archive name, and the open count for each file (0 means closed, greater than zero represents the number of times lib_open() has been called for the file).

If necessary, the index code can be changed while the method of using the librarian can be maintained. Changes to the global structures and to the librarian functions will definitely be required, but other code calling the librarian should be minimally affected, due to the basic functionality of the librarian primitives remaining the same.

The QUEST/Ada test data is read into a union type (lib_numeric_type) which is a joining of all of the integer and floating point types.

Some of the record types are "blocked", i.e., the data are broken into a number of individual, fixed-size records. This is due to some of the information stored in the temporary files are variable length. Part of the record's information is its block number. The define LIB_BLOCK_SIZE is used to decide how much information is allocated for each block. Also included in the record is a count for how many items in the block are used. If this count equals the LIB_BLOCK_SIZE, then the next block should be checked for existence. Once the count is less than the LIB_BLOCK_SIZE define, the last block in the data is reached.

5.2.5.4 BPLUS PORTABILITY NOTES

Much of the source code employed in the Librarian was originally intended for execution under MS-DOS. It was developed for the Microsoft C and the Borland Turbo C compilers. For the most part, standard C routines are employed for the file management. These routines, commonly known as the "UNIX" class of file routines, include open(), read(), write(), and close(). These routines should be standard in almost any implementation of a C compiler. Porting to the VAX required the deletion from the BPLUS.H and the BPLUS.C files of all instances of "cdecl" and of "Pascal". The #include statements had to be rearranged to either not include a file that did not exist on the VAX or to remove a "sys\" directory specification. Additionally, a filelength() function had to be written to allow the length of a file to be determined given the file's descriptor number. A phony #define for O_BINARY has been added so that an open() call succeeds. This binary specification is required for MS-DOS and other compilers that default to character translation for their data files.

An important note that might affect portability in the future has to do with the memcpy() function. In order for the code to run correctly on a Macintosh using the THINK C compiler, key memcpy() calls had to be changed to memmove(). This is because the ANSI standard of memcpy() now fails when overlapping memory space is involved. The function memmove() is specifically supposed to handle copying involving overlapping memory.

The BPLUS.H and BPLUS.C files contain function prototypes for the BPLUS functions. Only a compiler that contains the ANSI extensions to handle function prototypes can deal with their presence. Older style compilers (K&R vintage) will abort compilation

on encountering the function prototypes, requiring the declarations to be modified in order for the program to compile. Only the arguments contained within the prototype declaration need to be removed.

One final portability note is that the routine vsprintf() is called to print the ASCII representation of the key string (required for the BPLUS routines). This routine, although standard now, may not exist in older C libraries.

5.3 QUEST SYSTEM STRUCTURE

5.3.1 COMPONENT DIAGRAMS

The overall structure of the QUEST/Ada system was designed using the TAGS Input/Output Requirements Language (IORL). While the entire set of IORL specifications is given in the Phase 1 report [Brown89], some of these diagrams are updated in this section for illustration. Figure 5.3.1a shows the highest level of data flow, with the user interacting with the test environment, called QUEST (Query Utility Environment for Software Testing). Note that the IORL system description generally ignores the details of the user interface in order to simplify the model.

Table 5.3.1 presents the highest level data flows. As primary data flows, the user supplies source code and receives coverage analysis reports. Test cases are initially input by the user, who may continue to augment them throughout the test process. The user also interacts with QUEST to provide parameters to determine the extent and duration of testing. Requests for regression testing also proceed over interface QUEST-12. QUEST provides the means by which an execution of the module under test will produce output values for verification. Thus, actual module execution results also proceed over interface QUEST-21.

Figure 5.3.1b goes into more details of the QUEST system. The module being tested is input as Ada source code to the scanner/parser, which provides output to the test data generator (TDG), the test execution module (TEM), and the librarian (LIB). Figures 5.3.1c-f give a detailed graphical view of each of these modules. The interfaces between the various subsystems are described in the following section.

Table 5.3.1 Description of High Level Interfaces

INTERFACE	DESCRIPTION
QUEST-12	Source Code Test Data Generator Control Parameters Initial/Updated User Test Data Regression Test Signal
QUEST-21	Coverage Analysis Reports Source Code Listing Test Case Execution Results

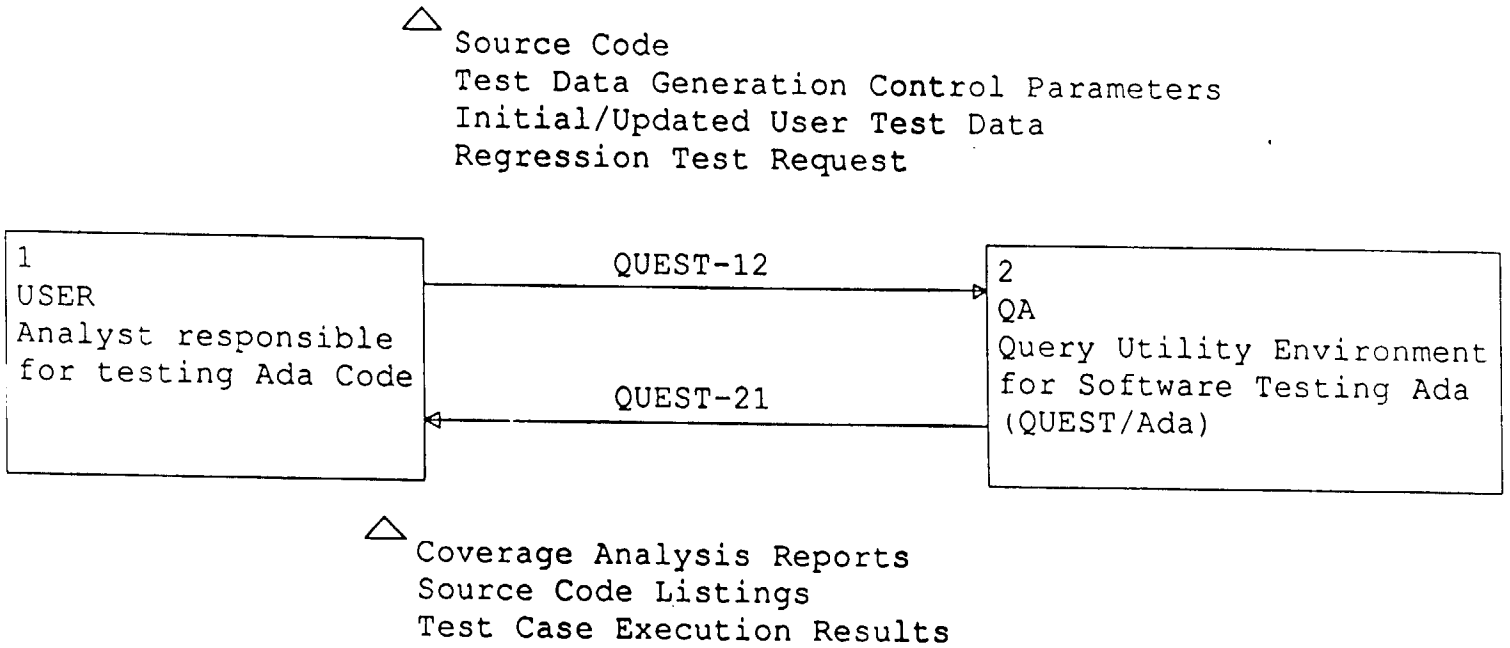
Source Code
Test Data Generation Control Parameters
Initial/Updated User Test Data
Regression Test Request
1
USER
Analyst responsible
for testing Ada Code
QUEST-12
QUEST-21
2
QA
Query Utility Environment
for Software Testing Ada
(QUEST/Ada)
Coverage Analysis Reports
Source Code Listings
Test Case Execution Results

Figure 5.3.1a

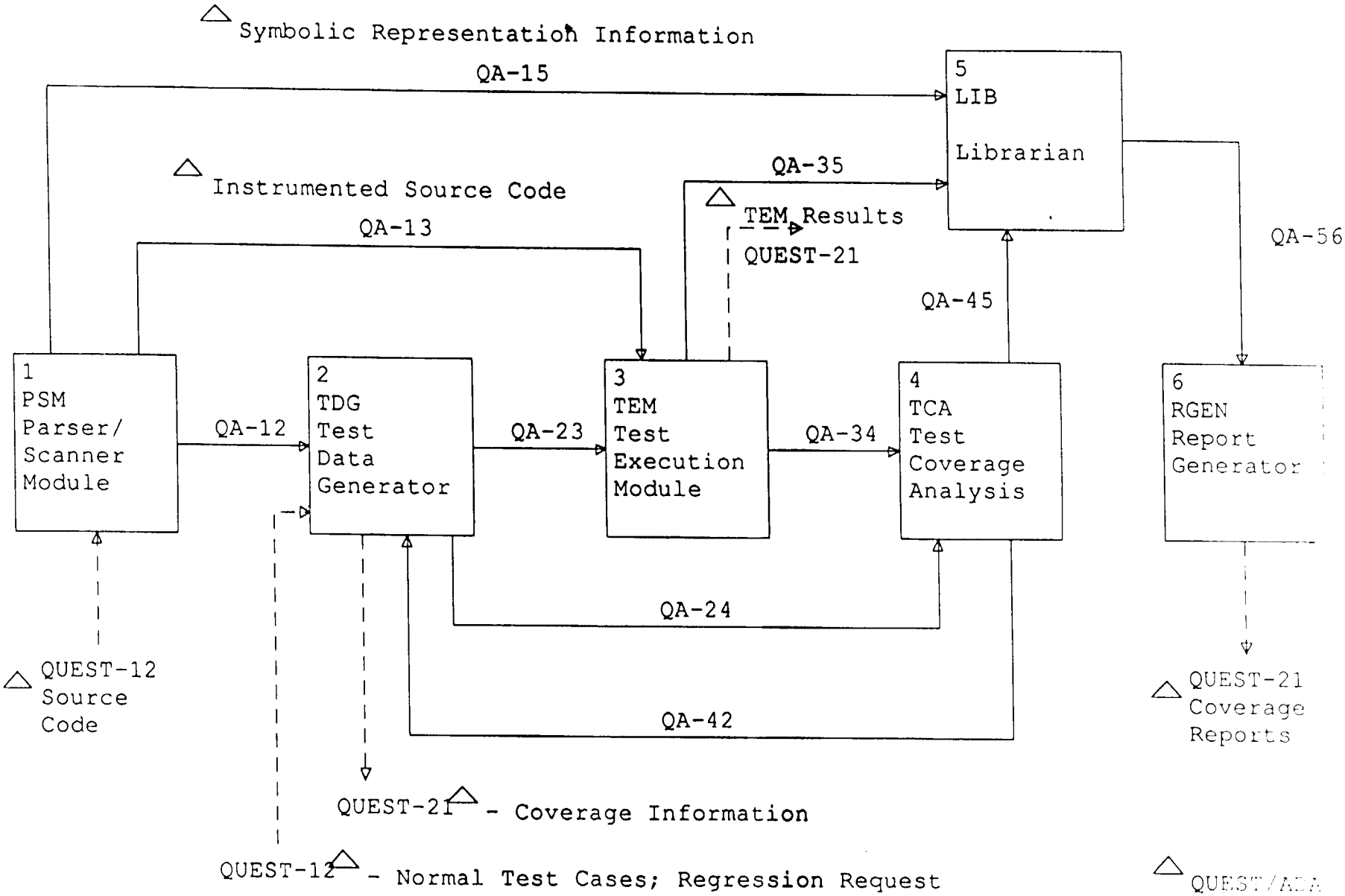
Symbolic Representation Information
QA-15
5
LIB
Librarian
Instrumented Source Code
QA-35
TEM Results
QUEST-21
QA-13
QA-56
QA-45
1
PSM
Parser/
Scanner
Module
QA-12
2
TDG
Test
Data
Generator
QA-23
3
TEM
Test
Execution
Module
QA-34
4
TCA
Test
Coverage
Analysis
6
RGEN
Report
Generator
QA-24
QUEST-12
Source
Code
QUEST-21
Coverage
Reports
QA-42
QUEST-21 - Coverage Information
QUEST-12 - Normal Test Cases; Regression Request
QUEST/ADA

Figure 5.3.1b

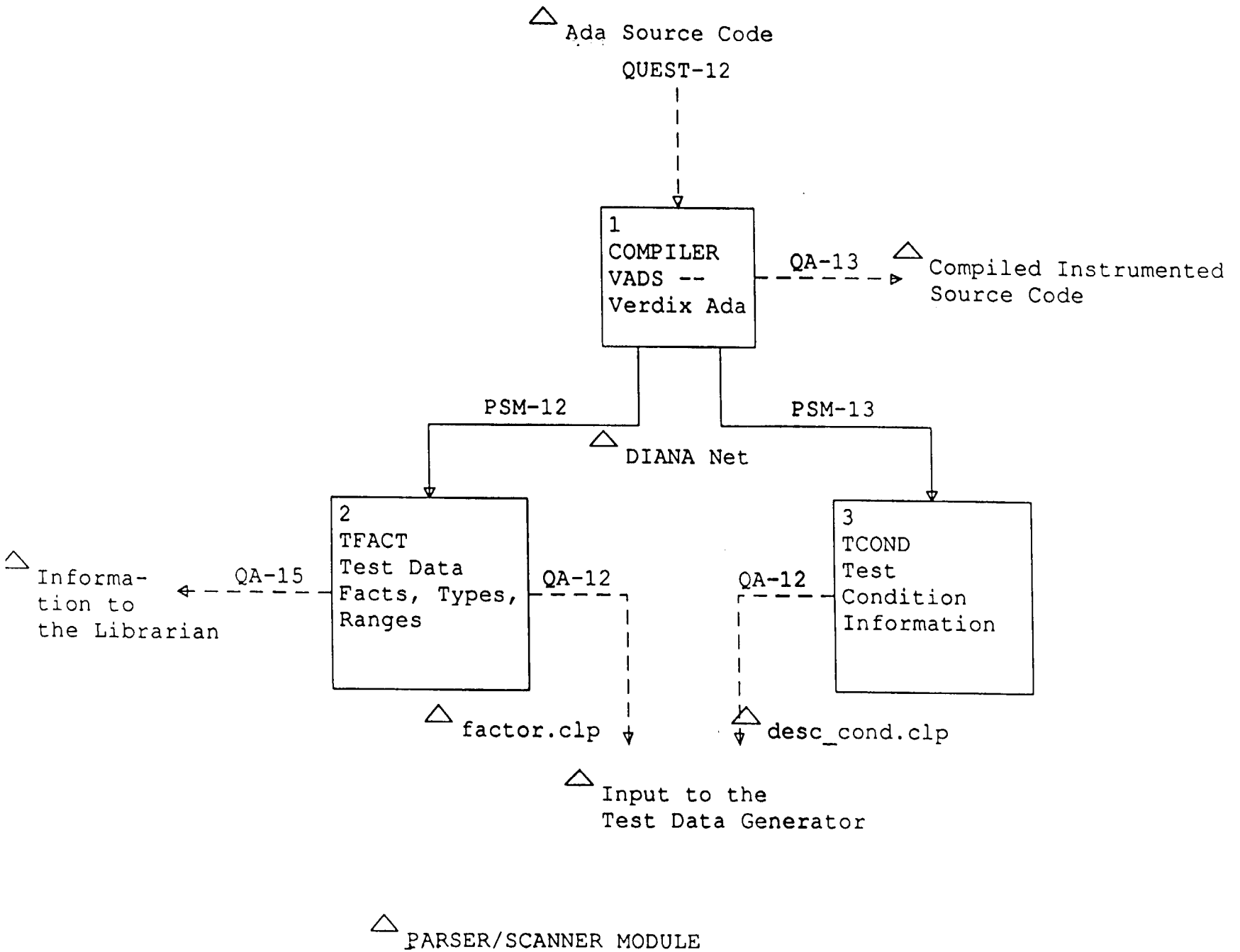
Ada Source Code
QUEST-12
1
COMPILER
VADS --
Verdix Ada
QA-13
Compiled Instrumented
Source Code
PSM-12
PSM-13
DIANA Net
2
TFACT
Test Data
Facts, Types,
Ranges
3
TCOND
Test
Condition
Information
QA-15
Informa-
tion to
the Librarian
QA-12
QA-12
factor.clp
desc_cond.clp
Input to the
Test Data Generator
PARSER/SCANNER MODULE

Figure 5.3.1c

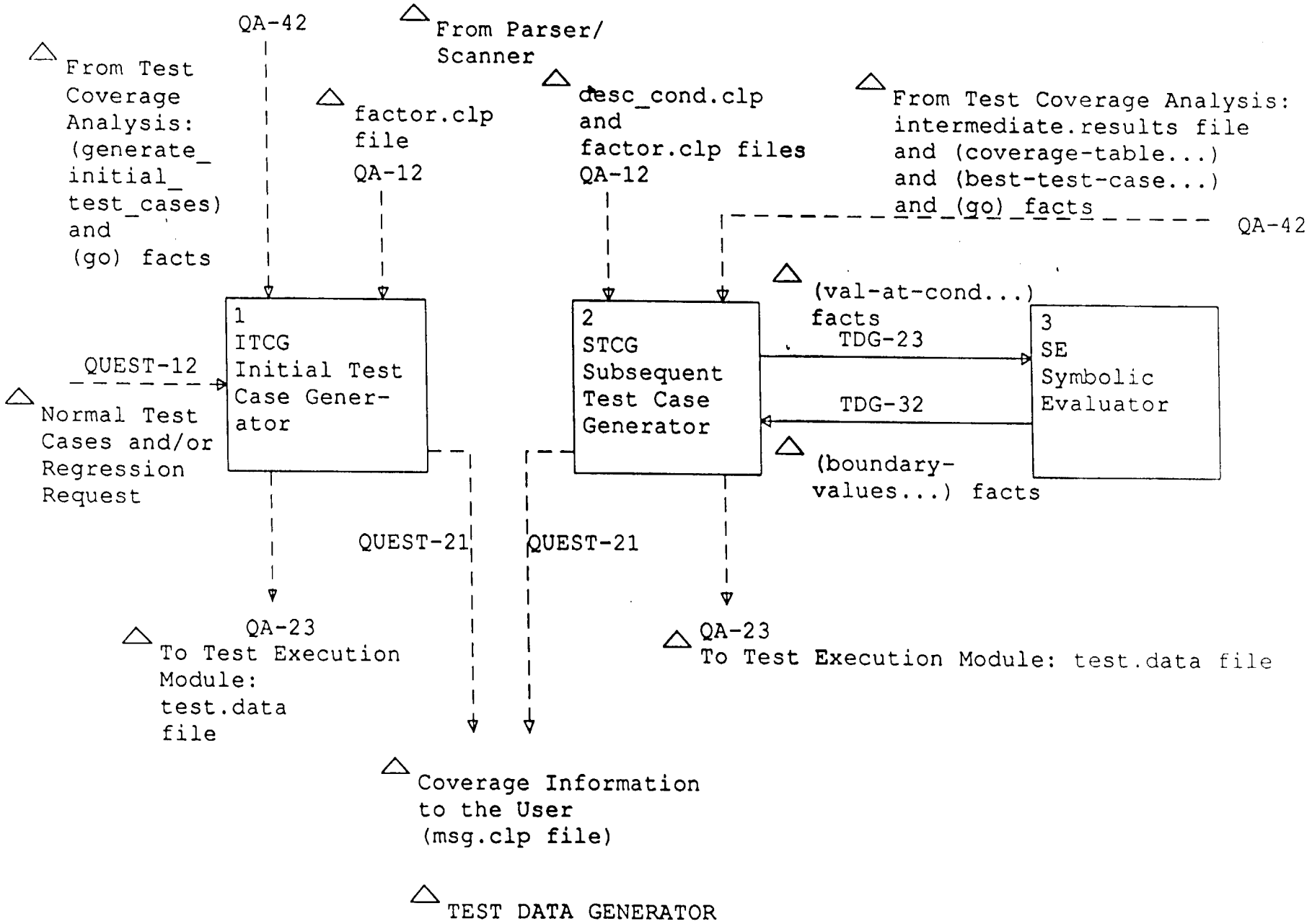
QA-42
From Parser/
Scanner
From Test
Coverage
Analysis:
(generate_
initial_
test_cases)
and
(go) facts
factor.clp
file
QA-12
desc_cond.clp
and
factor.clp files
QA-12
From Test Coverage Analysis:
intermediate.results file
and (coverage-table...)
and (best-test-case...)
and (go) facts
QA-42
(val-at-cond...)
facts
1
ITCG
Initial Test
Case Gener-
ator
2
STCG
Subsequent
Test Case
Generator
TDG-23
TDG-32
3
SE
Symbolic
Evaluator
QUEST-12
Normal Test
Cases and/or
Regression
Request
(boundary-
values...) facts
QUEST-21
QUEST-21
QA-23
To Test Execution
Module:
test.data
file
QA-23
To Test Execution Module: test.data file
Coverage Information
to the User
(msg.clp file)
TEST DATA GENERATOR

Figure 5.3.1d

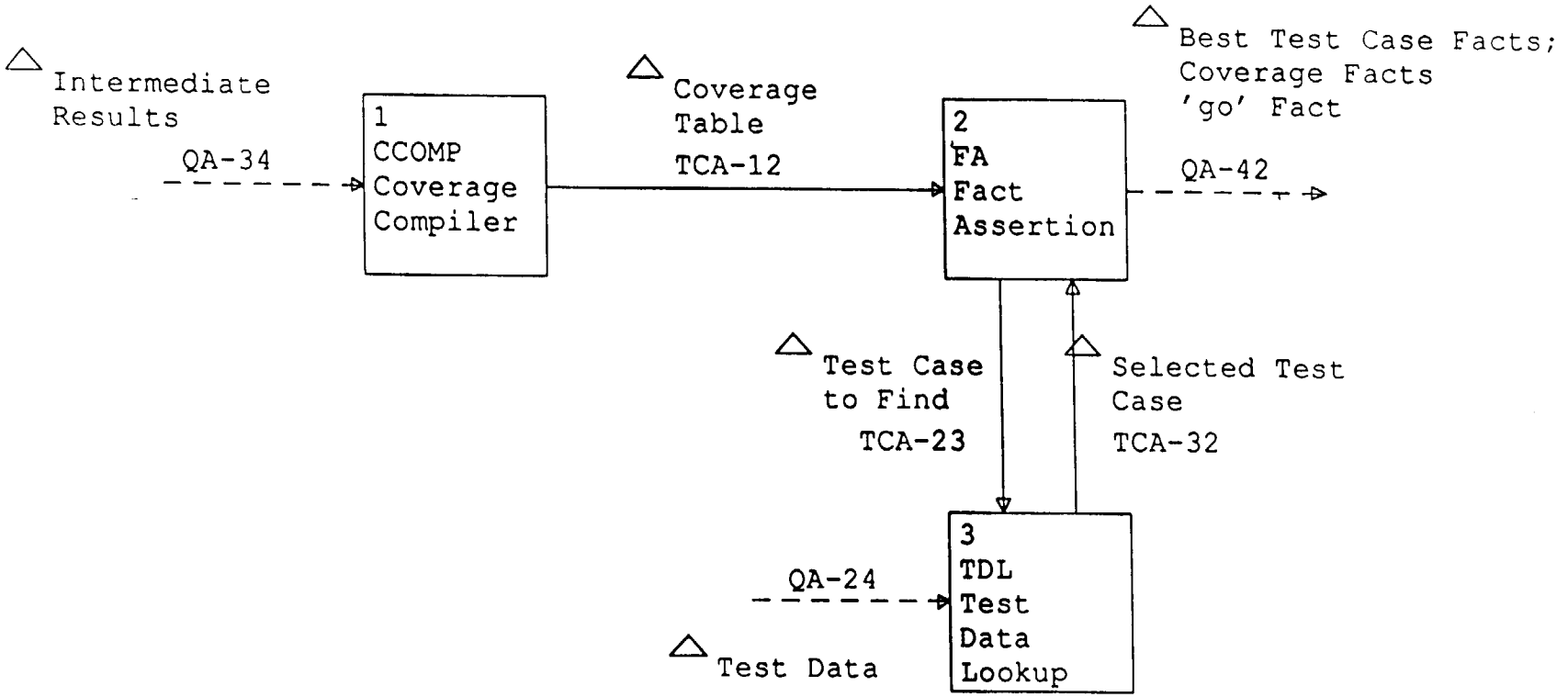
Intermediate Results
QA-34
1
CCOMP
Coverage
Compiler
Coverage Table
TCA-12
2
FA
Fact
Assertion
Best Test Case Facts;
Coverage Facts
'go' Fact
QA-42
Test Case to Find
TCA-23
Selected Test Case
TCA-32
3
TDL
Test
Data
Lookup
QA-24
Test Data
TEST COVERAGE ANALYSIS

Figure 5.3.1c

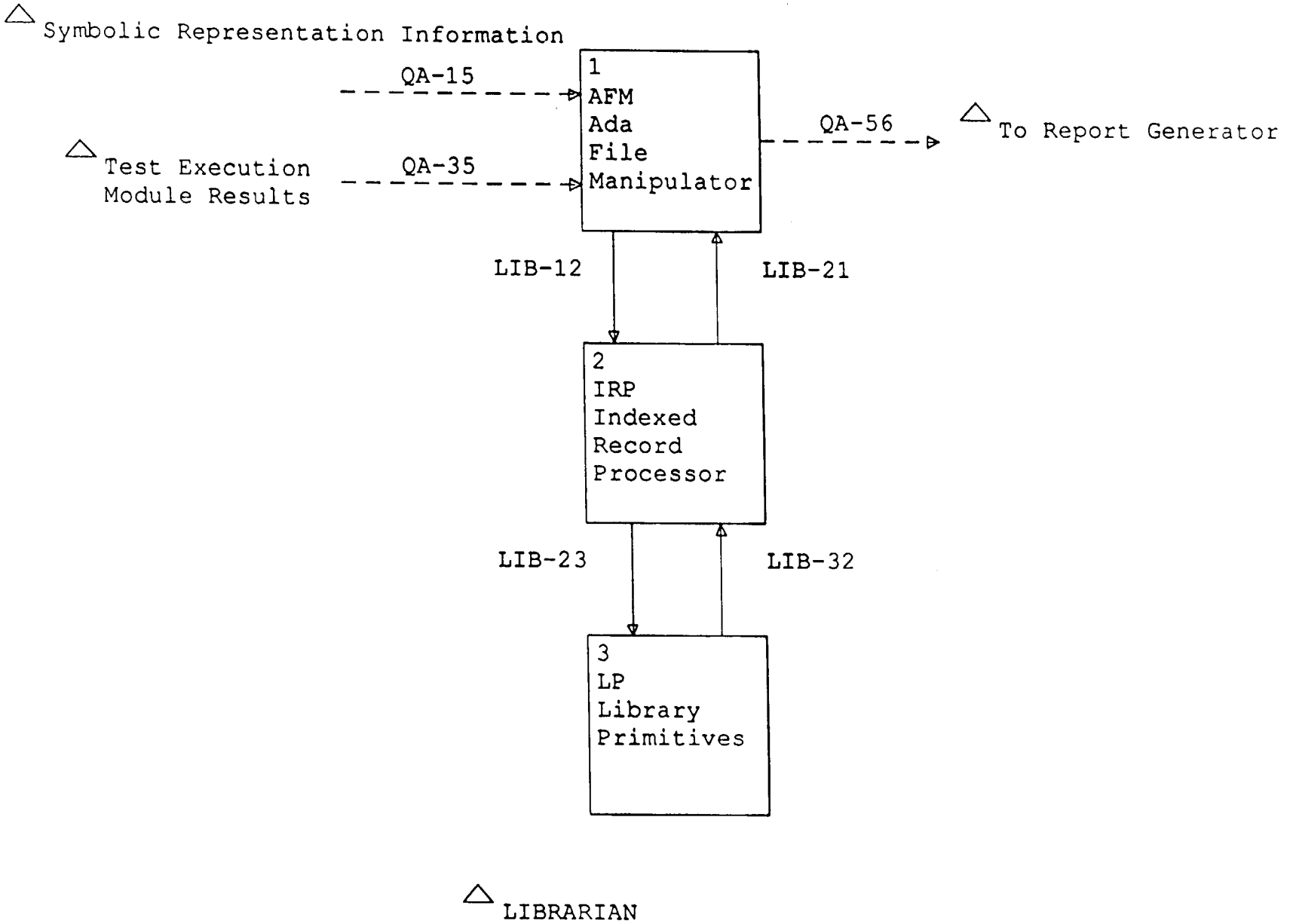
Symbolic Representation Information
QA-15
1
AFM
Ada
File
Manipulator
QA-56
To Report Generator
Test Execution
Module Results
QA-35
LIB-12
LIB-21
2
IRP
Indexed
Record
Processor
LIB-23
LIB-32
3
LP
Library
Primitives
LIBRARIAN

Figure 5.3.1f

5.3.2 DEFINITION OF HIGH LEVEL INTERFACES

5.3.2.1 PARSER/SCANNER INTERFACES

The parser/scanner produces data structures which describe the program under test to the test data generator and the report generator. This includes information concerning the input variables and parameters, condition and decision structure, and segment or block structure. The parser also augments the source code with a driver module for use by the test execution module. These interfaces are detailed in Table 5.3.2.1.

Table 5.3.2.1 PARSER/SCANNER MODULE INTERFACES

```
INPUT:     QUEST, ADA SOURCE CODE
               FROM: USER

OUTPUTS:   QA-13, INSTRUMENTED SOURCE CODE
                     TO: TEST EXECUTION MODULE
                   1. INSTRUMENTED DECISIONS
                   2. MODULE DRIVER

           QA-12, SYMBOLIC REPRESENTATION INFORMATION
               TO: TEST DATA GENERATOR
                   1. PARAMETER LIST
                   2. TYPE DECLARATIONS
                   3. DECISION/CONDITION DEFINITIONS
                        a. DECISION NUMBER
                        b. CONSTRUCT TYPE
                        c. DECISION STRUCTURE

           QA-15, SYMBOLIC REPRESENTATION INFORMATION
               TO: LIBRARIAN
                   1. DECISION/CONDITION LIST
                        a. DECISION NUMBER
                               b. CONSTRUCT TYPE
                               c. NUMBER OF CONDITIONS
```

5.3.2.2 TEST DATA GENERATOR INTERFACES

The Test Data Generator (TDG) interfaces are given in Table 5.3.2.2. The TDG obtains input from the parser/scanner in the form of a parse tree which describes the relevant structures within the source code. It translates this information into assertions which are used to determine the firing of the rule base.

The TDG interacts with the test execution module via test cases and test results. The results of each test case are analyzed by the Test Coverage Analyzer so that it can make decisions for the creation of additional test cases. This is performed by automatically analyzing the "quality" of the results generated at a given point in the testing process, where quality is determined by coverage metrics and variable value domain characteristics. The QA-23/QA-34/QA-42 loop is reiterated automatically until a given coverage is attained or until a user-defined check point is reached in terms of number of test cases generated. At this point the user will either stop the process or supply additional parametric information (via QUEST-12) to generate additional test data. User-defined test data may also be supplied at any of these check points.

Table 5.3.2.2 TEST DATA GENERATOR INTERFACES

```
INPUTS:    QUEST-12, TEST CASES:  NORMAL AND REGRESSION
               FROM:  USER

           QA-12, SYMBOLIC REPRESENTATION INFORMATION
               FROM: PARSER/SCANNER MODULE

           QA-42, TEST EXECUTION RESULTS
               FROM: TEST COVERAGE ANALYSIS


OUTPUTS:   QA-23, TEST CASES
               TO: TEST EXECUTION MODULE
                 1. TEST CASE NUMBER
                 2. TEST DATA

           QUEST-21, DYNAMIC COVERAGE INFORMATION
               TO: USER
```

5.3.2.3 TEST EXECUTION MODULE INTERFACES

The Test Execution Module (TEM) interfaces are shown in Table 5.3.2.3. TEM receives the instrumented source code sufficiently harnessed by a driver to enable it to be executed. Thus, its task is merely to execute the instrumented source code using as input the test data generated by the TDG component.

The TEM generates two outputs. The simplest of these is information for the Test Coverage Analysis (TCA). Each test case executed will produce an output via the instrumentation (i.e., a side effect) which will indicate the decision/condition satisfied by

that test case. This information will be processed by the TCA in order to serve appropriate information to the Test Data Generator and the Librarian.

The second output is a library of both the intermediate coverage information described above and the output results of each test case. This information will be stored for retrieval by the Regression Testing function and the Report Generator.

Table 5.3.2.3 TEST EXECUTION MODULE INTERFACES

```
INPUTS:      QA-13, INSTRUMENTED SOURCE CODE
                 FROM: PARSER/SCANNER MODULE

             QA-23, TEST CASES
                 FROM: TEST DATA GENERATOR


OUTPUTS:     QA-34, TEST EXECUTION RESULTS
                 TO: TEST COVERAGE ANALYZER
                      1. TEST CASE NUMBER
                      2. DECISION NUMBER
                      3. LIST OF VALUES OF DECISION VARIABLES
                   4. LIST OF CONDITION RESULTS

             QA-35, OUTPUT RESULTS
                 TO: LIBRARIAN

             QUEST-21, TEST CASE EXECUTION RESULTS
            TO:  USER
```

5.3.2.4 TEST COVERAGE ANALYSIS INTERFACES

Table 5.3.2.4 presents the Test Coverage Analyzer (TCA) interfaces. Essentially TCA takes the output generated via the probes inserted by the instrumentation and translates this information into the input required for efficient and straightforward report and test data generation. Note that this is accumulated in two formats, one for the analysis of an individual test case, and the other for the cumulative results of all tests performed. As mentioned above, a primary use of the former information is to provide feedback to the TDG to automatically generate improved test cases.

Table 5.3.2.4 TEST COVERAGE ANALYZER INTERFACES

```
INPUT:      QA-34, TEST EXECUTION COVERAGE RESULTS
                FROM: TEST EXECUTION MODULE

OUTPUTS:    QA-42, INTERIM COVERAGE ANALYSIS RESULTS
           TO: TEST DATA GENERATOR
                    1. TEST CASE NUMBER
                    2. DECISION NUMBER
                    3. LIST OF VALUES OF DECISION VARIABLES
                4. LIST OF CONDITION RESULTS

            QA-45, INTERMEDIATE COVERAGE ANALYSIS DATA
           TO: LIBRARIAN
                    1. INDIVIDUAL TEST COVERAGE DATA
                  a. TEST CASE NUMBER
                        b. DECISION NUMBER
                        c. CONDITION NUMBER
                        d. TRUE COUNT
                        e. FALSE COUNT
                    2. CUMULATIVE TEST COVERAGE DATA
                  a. DECISION NUMBER
                        b. CONDITION NUMBER
                        c. ACCUMULATIVE TRUE COUNT
                        d. ACCUMULATIVE FALSE COUNT
```

5.3.2.5 LIBRARIAN INTERFACES

The librarian serves as a mechanism for storing and retrieving information. It was described in detail in Section 5.2.5. The librarian interfaces are given in Table 5.3.2.5.

Table 5.3.2.5 Librarian Interfaces

```
INPUTS:     QA-45, INTERMEDIATE COVERAGE ANALYSIS DATA
           FROM: TEST COVERAGE ANALYZER

            QA-35, OUTPUT RESULTS
                FROM: TEST EXECUTION MODULE

            QA-15, SYMBOLIC REPRESENTATION INFORMATION
           FROM: PARSER/SCANNER MODULE

OUTPUTS:    QA-56, REPORT GENERATION INFORMATION
           TO: REPORT GENERATOR
```

5.3.2.6 REPORT GENERATOR INTERFACES

The symbolic representation information generated by the parser/scanner module is used in conjunction with the coverage measurements calculated by the coverage analysis module to produce detailed coverage analysis reports by the report generator. The user analyzes these reports to determine if there is a need for more tests. These interfaces are shown in Table 5.3.2.6.

Table 5.3.2.6 Report Generator Interfaces

```
INPUTS:      QA-56, REPORT GENERATION INFORMATION
            FROM: LIBRARIAN

OUTPUTS:     QUEST-21, TEST COVERAGE REPORTS
                 TO: USER
                     1. REPORT TYPES
                         a. INDIVIDUAL TEST COVERAGE
                   b. ACCUMULATIVE TEST COVERAGE
                2. COVERAGE TYPES
                         a. DECISION/CONDITION COVERAGE
                   b. MULTIPLE CONDITION COVERAGE
                   c. NO-HIT REPORT
```

5.4 DIRECTORY AND FILE DEFINITIONS

Basic directory structure of the Quest development directory:

DIRECTORY	DESCRIPTION
./progs/	Where test programs should go
./src/autotest	Autotest source code
./src/clips/clips4.1	Clips source code
./src/clips	Clips source code
./src/libr	Librarian source code
./src/xui/xquest	(Senior design project executable)
./src/xui	(Senior design project source)
./src/ada_packages	Misc. Ada packages
./src/misc	Misc. source code
./src/conc/sensor	Example tasking program source
./src/conc	Concurrency directory
./src	Source code for XQuest
./bin	Binary programs for XQuest
./rules	Clips rules (text)
./lib	Object code (Clips, librarian)
./includes	C include files (Clips, librarian)

./archives	Destination for archived information
./dumps	Various outputs
./results	Coverage reports
.	

Quest/includes

FILE NAME	DESCRIPTION
clips.h	CLIPS include file
constdef.h	CLIPS include file
librarian.h	Librarian include file
limits.h	CLIPS include file
quest.h	Librarian include file
structdef.h	CLIPS include file

Quest/lib

FILE NAME	DESCRIPTION
analysis.o	
bplus.o	Librarian: low level B-tree routines
clips.o	CLIPS routines
lib_quest.o	Librarian: high level archive routines
librarian.o	Librarian: mid level archive routines
main.o	CLIPS routines
math.o	CLIPS routines
npsr.o	CLIPS routines
parser.o	CLIPS routines
rulecomp.o	CLIPS routines
sysdep.o	CLIPS routines
sysfun.o	CLIPS routines
textpro.o	CLIPS routines
usrfun.o	CLIPS routines
usrint.o	CLIPS routines

Quest/rules

FILE NAME	DESCRIPTION
random.clp	CLIPS rule: random numbers
rdecby20p.clp	CLIPS rule: decrement by 20 percent
rdecby40p.clp	CLIPS rule: decrement by 40 percent
rdecby5.clp	CLIPS rule: decrement by 5
rinc_dec.clp	CLIPS rule: increment and decrement
se.clp	CLIPS rule: symbolic evaluator

Quest/src/autotest

FILE NAME	DESCRIPTION
KimberlysStuff	Directory -- unused
Makefile	Make file for creating xquest
README	Info file
README4-24-90	" "
README4-29-90	" "
README5-03-90	" "
SWindow.c	Modified HP widget private source file (scrolling window)
SWindow.h	Modified HP widget public include file
SWindowP.h	Modified HP widget private include file
accum_results2.c	Accumulate results
autotest.c	Autotest main loop
backup	Directory -- back ups
bldhlp	Program: build help file
bldhlp.c	Source code to bldhlp. Create help file
bldhlp.l	Lex source code to bldhlp
chckhlptb	Program: assists in building help file
chckhlptb.c	Source code to chckhlptb
help.c	Source code for help system
help.pre	Input help file
name_system.c	Source code used in setting up archive destination
other.c	X user interface control
project.c	X user interface control: project menu
quest.help.tab	Help input file
quest.help.text	Help input file
reports.c	X user interface control: reports menu
rptgen.c	Report generation routines
tcover.c	Coverage analysis
testing.c	X user interface control: test menu
xql.c	X user interface control
xql.h	X user interface control
xqmenus.c	X user interface control: menu utilities
xquest	Program: Quest/Ada
xquest.c	X user interface control: driver
xquest.h	X user interface control: header file

Quest/src/clips

FILE NAME	DESCRIPTION
README	Information file
analysis.c	CLIPS source code
clips.c	CLIPS source code
clips.exe Program:	CLIPS command line interface
clips.h	CLIPS source code
constdef.h	CLIPS source code
main.c	CLIPS source code
makefile	Make file for creating CLIPS
math.c	CLIPS source code
npsr.c	CLIPS source code
parser.c	CLIPS source code
rulecomp.c	CLIPS source code
structdef.h	CLIPS source code
sysdep.c	CLIPS source code
sysfun.c	CLIPS source code
textpro.c	CLIPS source code
usrfun.c	CLIPS source code
usrint.c	CLIPS source code

Quest/src/libr

FILE NAME	DESCRIPTION
README	Information file
bplus.c	Low level btree routines
bplus.h	Low level btree routines: include file
lib_def.h	Librarian definitions / data structures
lib_quest.c	Librarian high level routines
librarian.c	Librarian mid level routines
librarian.h	Librarian data structures

5.4 FILE DESCRIPTIONS

(To be completed in second half of Phase 3)

6.0 PROJECT SCHEDULE

6.1 SUMMARY OF PHASE 3 ACCOMPLISHMENTS

The goals of Task 1, Phase 3 are: (1) to further refine the rule base and complete the comparative rule base evaluation, (2) to implement and evaluate a concurrency testing prototype, (3) to convert the complete (unit-level and concurrency) testing prototype to a workstation environment, and (4) to provide a prototype development document to facilitate the transfer of the research technology to a working environment. The progress in achieving these goals will now be discussed by subtask.

1. Refinement of the rule base.

The symbolic evaluation rule base developed in Phase 2 was capable of generating test data for unit-level testing which obtains coverage that is very difficult to achieve with standard test case generation. However, continued enhancements of the rule base are necessary in order for this technique to reach its full potential. The objective is to generate test sets which either lead to greater coverage or to the same coverage with fewer test cases generated. Refinements to the symbolic evaluation rule base have been implemented using the CLIPS expert system tool in an attempt to implement the design work that was done in Phase 2. Two measurements were designed to select the best test case. One is based solely on the target condition while the other is based on the target condition plus the conditions that are on the path to the target condition. During the second phase, one of the three developed methods is selected to generate new cases based on the given best case. The three methods of modification are random, fixed percentage, and symbolic evaluation. While some preliminary tests of this new scheme have been generated, work will continue in the evaluation and refinement of the rule base during the remainder of the project.

2. Completion of the rule base evaluation.

The comparative evaluation of the various rule-based testing strategies has been continued based upon the preliminary tests described above. Actual execution of the updated prototype enabled the comparison of the test case generation rules with those previously applied, both for the example modules which were previously used to test QUEST, and some of the NASA-supplied modules. The completed evaluations are mixed with regard to the advantages of the new rule base, and they will be further used to qualitatively assess the strengths and weaknesses of each rule and rule type. Some showed the newly implemented approach to be quite good, while others showed them to be about the same as previous methods. Since these were performed late in the reporting period, a detailed analysis of these results has not been performed. These results will be invaluable in guiding the remaining part of the project. An attempt will also be made to quantify the effects of interacting rule bases on the unit-level coverage of a variety of Ada programs. Work will also continue in an attempt to identify an optimal general rule base across a cross-section of testing paradigms and programs.

3. Implementation and evaluation of a concurrency testing prototype.

The work on concurrency construct testing was confined to some additional design work to further evaluate the alternatives proposed at the end of Phase 2. It has been determined that the lock-step/monitor approach has the greatest chance for success. In the remainder of the project the design will be further refined and implemented using concurrency tasking information provided by the Verdix DIANA Ada interface package. The concurrency testing approach will provide current history coverage information through the use of the "iron-fist" task scheduling monitor which will force determinism into the Ada rendezvous by locking all but one possible rendezvous in the case of multiple rendezvous selects. It is anticipated that this approach, combined with the unit-level testing approach already developed in Phases 1 and 2, will initially cover all rendezvous. The rendezvous coverage prototype will be evaluated to determine the possibilities of extending the coverage metric to a more general case, such as Taylor task histories.

4. Development of a workstation-environment prototype.

The meeting conducted with Boeing during Phase 2 of the project indicated the need for a workstation environment for the testing prototype. This workstation environment has now been developed (in particular, Sun SPARCstation, UNIX, XWindows). In addition to providing a new user interface which reflects current user interface design techniques, this development also affords the opportunity to expand the features of the prototype. One important expanded feature is the use of the Verdix DIANA Ada interface package in the place of the previous attributed grammar in the parser/scanner module. It is projected that the use of the DIANA interface will also provide advantages in the development of the concurrency prototype, aid the transition from prototype to working package, and make QUEST compatible with the APSE standard. Additional work needs to be done, however, in enabling the prototype to take full advantage of this environment. In particular, the ability to execute the module under test without leaving the QUEST environment is a feature which will be added along with other improvements in the user interface during the remainder of Phase 3.

Work within this activity also included the overall upgrade of the QUEST prototype environment. To a large extent developments within this period with regard to refining the prototype have been driven by the example Ada code modules which were obtained from NASA. Since the previous version of the prototype had only considered integer types, fixed and floating point types were added to the test data generation rules. Also, provisions were made to handle multiple conditions and global variables.

5. Development of a technology-transfer document.

In order to speed the transfer of technology from the research environment to a working environment, a preliminary development document has been drafted. While certain significant sections are not completed, the document provides the basis for a concise overview and a detailed explanation of each module of the prototype system. In future versions, directions which might be taken to expand the prototype modules into a more robust system will be added. The purpose of this document will be to allow any interested

NASA subcontractor to quickly develop a robust working automated testing environment from the prototype developed during this research.

6. Continue contacts with NASA subcontractors currently developing Ada software.

The contacts established with Boeing in Phase 2 of this project provided useful insight into the requirements NASA subcontractors have for an automated program testing tool. Continued interaction with these contacts will be sought in the development of a concurrency testing prototype appropriate to existing concurrent Ada software. Contacts have also been made with Science Application International Corporation (SAIC), Optimization Technology Incorporated (OTI), and McCabe Associates in order to arrange joint agreements for the further development and eventual marketing of the QUEST system.

6.2 PROPOSED RESEARCH SCHEDULE

The Gantt chart in Figure 6.2 provides the sequence of Task I activities to be accomplished during Phase 3 of this project as presented above. Progress in completing Phase 3 has been delayed somewhat by the lack of available graduate students and the delayed arrival of some important software components. This problem has been addressed by additional recruiting and the receipt of the software. However, a no-cost extension to September 31, 1991 has been be requested, and this is anticipated to be adequate for completing all of the activities scheduled.

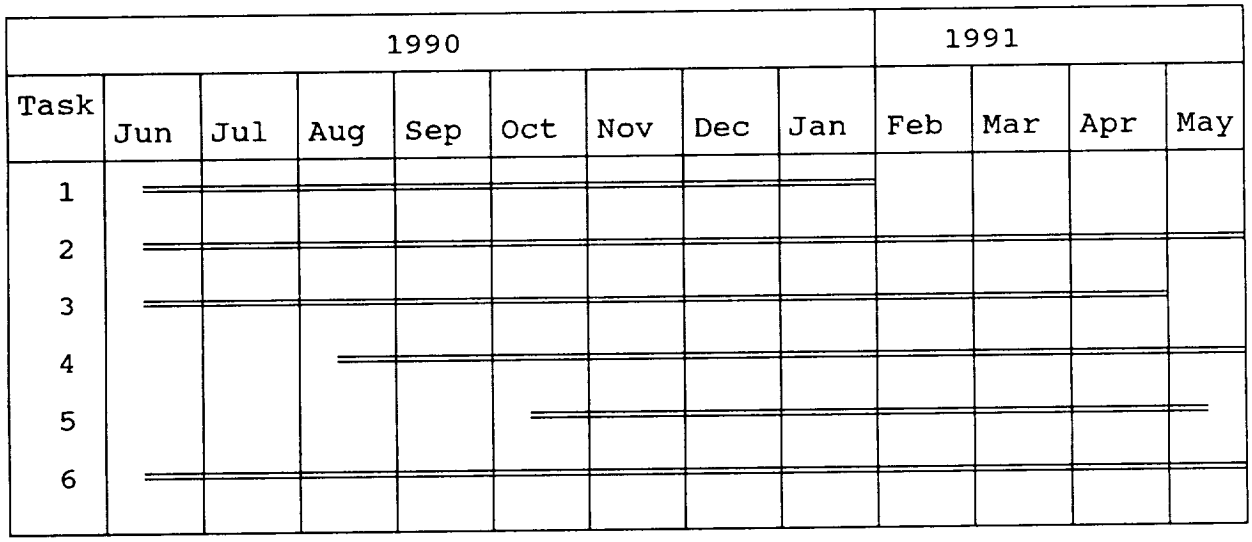

Figure 6.2 Phase 3, Task I Gantt Chart

7.0 BIBLIOGRAPHY

[Brin89] Brindle, A. F., Taylor, R. N., and Jansen, L. R., "A Debugger for Ada Tasking", **IEEE Transactions on Software Engineering**, Vol. 15, No. 3, pp. 293-304, March 1989.

[Brown89] Brown, D. B., "QUEST/Ada: The Development of a Program Analysis Environment for Ada," Phase 1 Report, NASA Contract Number NASA-NCC8-14, June, 1989.

[Brown90] Brown, D. B., et al "QUEST/Ada: The Development of a Program Analysis Environment for Ada," Task 1, Phase 2 Report, NASA Contract Number NASA-NCC8-14, August, 1990.

[Call89] Callahan, D. and Subhlok, J., "Static Analysis of Low-level Synchronization", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 100-111, January 1989.

[Chen87] Cheng, J., Araki, K., and Ushijima, Kazuo, "Event-driven Execution Monitor for Ada Tasking Programs", **Proceedings of COMPSAC 87**, pp. 381-388, October 1987.

[Cli87] CLIPS Reference Manual, Version 4.1, Artificial Intelligence Section, Johnson Space Center, NASA, September 1987.

[Cross89] J.H. Cross II, K.I. Morrison, C.H. May, and K.C. Waddel, A Graphically Oriented Specification Language for Automatic Code Generation (phase 1), NASA Annual Report, Department of Computer Science and Engineering, Auburn University, August 1989.

[Dea88] W. H. Deason, "Rule-based Software Test Case Generation,' M.S. Thesis, Department of Computer Science and Engineering, December 1988.

[Dea91] W. H. Deason, D. B. Brown, K.-H. Chang, and J. H. Cross II, "A Rule-based Software Test Data Generator," IEEE Trans. on Knowledge and Data Engineering, March 1991. (To appear)

[Dian90] DIANA Interface Package Manual, Verdix Corp., CA. 1990.

[Dil90] Laura K. Dillon, "verifying Safety Properties of Ada Tasking Programs," IEEE Transactions on Software Engineering, Vol.. 16, No. 1, January 1990, pp. 51 - 63.

[DeM78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," IEEE Computer, Vol. 11, No. 4, April 1978.

[Fali82] Falis, Edward, "Design and Implementation in Ada of a Runtime Task Supervisor", **Proceedings of the AdaTEC Conference on Ada, published in ACM SIGPLAN Notices**, pp. 1-9, October 1982.

[Fran88] Franscesco, N. D. and Vaglini, G., "Description of a Tool for Specifying and Prototyping Concurrent Programs", **IEEE Transactions on Software Engineering**, Vol. 14, No. 11, pp. 1554-1564, November 1988.

[Gait86] Gait, J., "A Probe Effect in Concurrent Programs", **Software -Practice and Experience**, Vol. 16, No. 3, pp. 225-233, March 1986.

[Germ82] German, S. M., Helmbold, D. P., and Luckham, D. C., "Monitoring for Deadlocks in Ada Tasking", **Proceedings of the AdaTEC Conference on Ada, published in ACM SIGPLAN Notices**, pp. 10-27, October 1982.

[Germ84] German, S. M., "Monitoring for Deadlock and Blocking in Ada Tasking", **IEEE Transactions on Software Engineering**, Vol. SE-10, No. 6, pp. 764-777, November 1987.

[Gold89] Goldszmidt, G. S., Katz, S., and Yemini, S., "Interactive Blackbox Debugging for Concurrent Languages", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 271-283, January 1989.

[Hail82] Hailpern, B. T., **Verifying Concurrent Processes Using Temporal Logic**, Springer-Verlag, Berlin, 1982.

[Helm85] Helmbold, D. and Luckham, D., "Debugging Ada Tasking Programs", **IEEE Software**, Vol. 2, No. 2, pp. 47-57, March 1985.

[Helm85b] Helmbold, D. and Luckham, D. C., "Runtime Detection and Description of Deadness Errors in Ada Tasking", **Ada Letters**, Vol. 4, No. 6, pp. 60-72, 1985.

[How86] W.E. Howden, "A Functional Approach to Program Testing and Analysis," IEEE Trans. on Software Engineering, Vol. SE-12, No. 10, October 1986.

[Hseu89] Hsuesh, W. and Daiser, G. E., "Data Path Debugging: Data-oriented Debugging for a Concurrent Programming Language", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 236-247, January 1989.

[Lamp89] Lamport, L., "A Simple Approach to Specifying Concurrent Systems", **Communications of the ACM**, Vol. 32, No. 1, pp. 32-45, January 1989.

[LeDou85] LeDoux, C. H. and Parker, D. S., "Saving Traces for Ada Debugging", **Ada in Use, Proceedings of the Ada International Conference, published in ACM Ada Letters**, Vol. 5, No. 2, pp. 97-108, September 1985.

[Lewis84] Lewis, T. G., Spitz, K. R., and McKenney, P. E., "An Interleave Principle for Demonstrating Concurrent Programs", **IEEE Software**, Vol. 1, No. 4, pp. 54-64, October 1984.

[Mill88] Miller, B. P. and Choi, J.D., "A Mechanism for Efficient Debugging of Parallel Programs", **Proceedings of the SIGPLAN '88 Conference on Programming Language and Implementation, published in ACM SIGPLAN Notices**, Vol. 23, No. 7, pp. 135-144, July 1988.

[Mura89] Murata, T., Shender, B., and Shatz, S. M., "Detection of Ada Static Deadlocks Using Petri Net Invariants", **IEEE Transactions on Software Engineering**, Vol. 15, No. 3, pp. 314-325, March 1989.

[Pra87] R.E. Prather and P. Myers, Jr., "The Path Prefix Software Testing Strategy," IEEE Trans. on Software Engineering, Vol. SE-13, No. 7, July 1987.

[Ston88] Stone, J. M., "Debugging Concurrent Processes: a Case Study", **Proceedings of the SIGPLAN '88 Conference on Programming Language and Implementation, published in ACM SIGPLAN Notices**, Vol. 23, No. 7, pp. 145-153, July 1988.

[Ston89] Stone, J. M., "A Graphical Representation of Concurrent Processes", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 226-235, January 1989.

[Stran81] Stranstrup, J., "Analysis of Concurrent Algorithms", **Lecture Notes in Computer Science Conpar 81 (Conference on Analyzing Problem Classes and Programming for Parallel Computing)**, pp. 217-230, June 1981.

[Tai85] Tai, K. C., "On Testing Concurrent Programs", **Proceedings of COMPSAC 85**, pp. 310-317, October 1985.

[Tai86] Tai, K. C., "A Graphical Notation for Describing Executions of Concurrent Ada Programs", **ACM Ada Letters**, Vol. 6, No. 1, pp. 94-103, January 1986.

[Tayl78] Taylor, R. N. and Osterweil, L. J., "A Facility for Verification, Testing, and Documentation of Concurrent Process Software", **Proceedings of COMPSAC 78**, pp. 36-41, November 1978.

[Tayl80] Taylor, R. N. and Osterweil, L. J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", **IEEE Transactions on Software Engineering**, Vol. SE-6, No. 3, pp. 265-277, May 1980.

[Tayl83a] Taylor, R. N., "Complexity of Analyzing the Synchronization Structure of Concurrent Programs", **Acta Informatica**, Vol. 19, pp. 57-84, April 1983.

[Tayl83b] Taylor, R. N., "A General-Purpose Algorithm for Analyzing Concurrent Programs", **Communications of the ACM**, Vol. 26, No. 5, pp. 362-376, May 1983.

[Tayl88] Taylor, R. N. and Young, M., "Combining Static Concurrency Analysis with Symbolic Execution", **IEEE Transactions on Software Engineering**, Vol. 14, No. 10, pp. 1499-1511, October 1988.

[TBE84] Teledyne Brown Engineering, IORL -- Input/Output Requirements Language Reference Manual, Teledyne Brown Engineering, Cummings Research Park, Huntsville, Alabama 35807, July, 1984.

[Utte89] Utter, P. S., and Pancake, C. M., "A Bibliography of Parallel Debuggers", **SIGPLAN Notices**, Vol. 24, No. 11, pp. 29-42, Nov., 1989.