QUEST/Ada

QUERY UTILITY ENVIRONMENT FOR SOFTWARE TESTING OF ADA

The Development of a Program Analysis Environment for Ada

Contract Number NASA-NCC8-14

Phase 1 Report

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

Contact: David B. Brown, Ph.D., P.E.
Professor and Interim Head
(205) 844-4330
dbrown@AUDUCVAX.bitnet

June 1, 1989

TABLE OF CONTENTS

Acknowledgements

Executive Summary

1. Introduction 1

2. Literature Review 1
   - 2.1 Introduction 1
   - 2.2 Software Testing 2
     - 2.2.1 Functional Testing 2
     - 2.2.2 Structural Testing 3
     - 2.2.3 Need For Both Functional and Structural Testing 4
     - 2.2.4 Other Test Strategies 4
       - 2.2.4.1 Mutation Testing 4
       - 2.2.4.2 Domain Testing 5
       - 2.2.4.3 Symbolic Evaluation 5
   - 2.3 Automation 6
     - 2.3.1 The Need For An Oracle 6
     - 2.3.2 Automated Testing Tools 6
       - 2.3.2.1 Structural Testing Tools 6
       - 2.3.2.2 Functional Testing Tools 7
   - 2.4 Reliability Models and Test Adequacy Criteria 7
   - 2.5 Test Data Generation 8
   - 2.6 The Path/Predicate Solution Problems 8
   - 2.7 Conclusion 9

3. Definition of System Structure 10

4. Definition of High-Level Interfaces 13
   - 4.1 Parser/Scanner Interfaces 13
   - 4.2 Test Data Generator Interfaces 14
   - 4.3 Test Execution Module Interfaces 15
   - 4.4 Test Coverage Analysis Interfaces 16
   - 4.5 Report Generator Interfaces 17

5. Scope of the Prototype 18

6. Parser/Scanner Design 19
   - 6.1 General Parser/Scanner Requirements 19
   - 6.2 Parser/Scanner Overview 19
   - 6.3 Limitations of the Parser/Scanner Prototype 20
   - 6.4 Example Module Instrumentation 21
   - 6.5 Parser/Scanner Implementation Details 27
   - 6.6 Future Parser/Scanner Research Efforts 30

7. Test Data Generator Design 32
   - 7.1 Initial Case Generation 34
   - 7.2 Coverage Analysis and Goodness Evaluation 34
   - 7.3 New Test Case Generation 36

7.4 Test Data Generation Examples 38
7.5 System Interface Mechanism 40

8. Test Coverage Analyzer 42

9. Recommendations for Future Research 43

10. References 44

APPENDIX A - QUEST/Ada IORL System Specifications

APPENDIX B - Paper: A Rule-Based Software Test Data Generator

APPENDIX C - Paper: QUEST/Ada: An Automated Tool for Ada Software Testing

APPENDIX D - Paper: Expert System Assisted Test Data Generation for Software Branch Coverage

APPENDIX E - Design of the User Interface
E.1 System Definition Menu
E.2 Module Selection Menu
E.3 Automatic Testing Menu
E.4 QUEST Regression Test Menu
E.5 QUEST Variable Definition Menu
E.6 Testing Result Reports Menu
E.7 QUEST Utilities Menu
E.8 Detailed Plan for Project

APPENDIX F - Proposal for Phase II

ACKNOWLEDGEMENTS

Portions of this report were contributed 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. Homer W. Carlisle
Dr. Kai-Hsiung Chang
Dr. James H. Cross

GRADUATE RESEARCH ASSISTANTS

William H. Deason
Kevin D. Haga
John R. Huggins
William R. A. Keleher, Technical Manager
Benjamin B. Starke
Orville R. Weyrich
Michael P. Woods

UNDERGRADUATE TECHNICAL ASSISTANTS

Todd E. Blevins
J. Edward Swan
David Wheeler

Ada is a trademark of the United States Government, Ada Joint Program Office.

EXECUTIVE SUMMARY
THE DEVELOPMENT OF A PROGRAM ANALYSIS ENVIRONMENT FOR ADA

After several preliminary meetings with the sponsor, the scope of this project was defined to include the design and development of a prototype system for testing Ada software modules at the unit level. This would be patterned after a previous prototype for FORTRAN developed at Auburn University. The new system was called Query Utility Environment for Software Testing of Ada (QUEST/Ada).

QUEST/Ada differs significantly from its predecessor in the following regard: (1) the parser/scanner mechanism is obtained from a formal parser/scanner generator such as YACC, LALR 3.0, or BISON, (2) the test data generator is rule-based as opposed to traditional techniques of path generation and predicate solution, and (3) a large number of test cases are assumed to be supportable. This third difference assumes the presence of redundant code generated either automatically from the specification (sometimes called simulation) or by manual coding. With automatic comparison capabilities there is no longer a need for selecting only a relatively few test cases for verification. QUEST/Ada is being designed under the premise that a large number of test cases will be generated from the rule base. A subset of these, which provide the necessary path and domain coverage characteristics, may be selected for verification.

The literature review can be summarized by a quotation from Fisher which stated that currently "there are no CASE tools to assist in the unit test and integration phase" [FIS88]. However, the literature abounds with papers on the theory of software testing, and much work is continuing in this area. The literature review was organized according to: (1) software testing approaches and strategies, (2) automation of the various aspects of software testing, (3) reliability models and test adequacy criteria, (4) test data generation approaches, and (5) a discussion of rule-based versus traditional test data generation approaches.

The design of QUEST/Ada began with a definition of the overall system structure. This was performed in IORL, which tended to clarify component dependencies for the project team. This led to a more formal description of these dependencies, which was obtained by the definition of the high level interfaces between the components. The project team was then subdivided into three groups to resolve the preliminary design of the major three components of QUEST/Ada, namely: (1) the parser/scanner, (2) the test data generator, and (3) the test coverage analyzer.

The Phase I report is organized as a working document from which the system documentation will evolve. The introductory section provides some history and a guide to the sections of the report. A fairly comprehensive literature review follows which is targeted toward issues of Ada testing. The definition of the system structure and the high level interfaces are then presented. A brief description of the scope of the prototype is given next. This is followed by a chapter on the design of each of the three major components. Finally, the plan for the remainder of the project is given. The appendices include the QUEST/Ada IORL System Specifications to this point in time, a paper presenting statistical evidence of the validity of the rule-based testing paradigm, and two other papers derived from the current research on QUEST/Ada. In addition, a preliminary users' manual for the current QUEST user interface is provided. The final appendix includes the proposal and workplan for Phase II.

1. INTRODUCTION

This project was initiated on June 1, 1988. Because funding of the original proposal was reduced, the Principal Investigator and the NASA representatives spent the major portion of the first month defining the scope of the project. A meeting was held on July 1, 1988 at Auburn to present and verify this redefinition. Generally the project was subdivided with a minor pilot effort being devoted toward an analysis of metrics for the evaluation of existing software packages. Dr. Cherri Pancake and a graduate student were assigned to this component of the project, and the results of their efforts are presented in a separate report.

The meeting on July 1, 1988 resolved that the major emphasis of the project would be in the direction of the design and prototyping of an environment to facilitate the testing of Ada code. This would be modeled after an available prototype environment for FORTRAN code testing, called QUEST. However, several new approaches were required in order to enable Ada code to be tested. Among these were: (1) the use of a formal grammar to generate the parser to be used in the prototype, (2) the use of rule-based techniques for generating test cases, and (3) the ultimate development of testing approaches to handle concurrency. The first two of these were considered in the current project.

A second meeting was held on October 6, 1988 in Huntsville in which the progress over the first three months of the project was reported. This included results of: (1) the literature review (2) a definition of overall system structure, (3) a definition of high level interfaces, (4) a definition of the Ada subset to be processed by the prototype, (5) a preliminary analysis of scanner/parser requirements, and (6) a detailed plan for the second quarter.

This report continues by presenting the results of the literature review which clearly reveals a gap in the area of automatic test data generation for Ada unit-level testing. This is followed by the definition of the QUEST/Ada system structure, which shows a high-level view of the components of the system. A definition of the high level interfaces is then presented which tends to further crystallize the component design. In Section 5 the Ada scope of the prototype under development is defined. This is followed by the definition of parser/scanner design, which contains an example module instrumented by an early prototype. Section 7 presents the rule-based test data generator design, and Section 8 presents the test coverage analyzer design, after which the plan for the next phase of the project is given. Finally, the high level IORL description of QUEST/Ada, the preliminary QUEST users' manual, three related papers, and the proposal for Phase II are given in the Appendices.

2. LITERATURE REVIEW

2.1 INTRODUCTION

With the increased production of complex software systems for embedded systems applications, it becomes apparent that without some form of organized and efficient approach to the design, development and testing phases of the software lifecycle, software reliability for these systems will fall short of the goals set by their developers. A variety of approaches to software testing exist [ADR82, GOO75, HOW80, HOW76, HOW82a, WHI80]. However, these methodologies generally require considerable manual effort, i.e., the tester must hand compute paths, predicates, test cases, etc. Manual implementation of these methodologies is not only inefficient in terms of resources expended (man-hours), but it is also subject to inconsistencies brought about by human errors. Manual methods can generate only a

limited number of test cases before the amount of time expended becomes unacceptably large. All of these problems may be reduced by the use of automated software test tools. However, automated test data generation itself is not well understood [MIL 84, PAN 78].

Ramamoorthy defines automated test tools ".. as programs that check the presence of certain software attributes which can be program syntax correctness, proper program control structures, proper module interface, testing completeness, etc." [RAM75]. This is the goal of the QUEST/Ada testing tool: to reduce the resources that must be expended by automating portions of the testing phase previously requiring manual intervention. Currently "there are no CASE tools to assist in the unit test and integration phase" [FIS88].

2.2 SOFTWARE TESTING

Software testing as a software engineering discipline is coming of age in the 80's. As E. F. Miller pointed out [MIL84], "there is growing agreement on the role of testing as a software quality assurance discipline, as well as on the terminology, technology, and phenomenology of, and expectation about testing." He also noted that the first formal conference on software testing took place at the University of North Carolina in June of 1972. Since that time, testing research has continued on several fronts, including the automation of portions of the testing process.

In the testing stage of the software life cycle, the main thrust of research has been aimed at developing more formal methods of software and system testing [BEI83]. By definition, "testing...is the process of executing a program (or a part of a program) with the intention or goal of finding errors" [SHO83]. A test case is a formally produced collection of prepared inputs, predicted outputs, and observed results of one execution of a program [BEI83]. In standard IEEE terminology, a software fault is an incorrect program component; an error is an incorrect output resulting from a fault. In order to detect occurrences of errors indicating faults, some external source of information about the program under test must be present.

Program testing methods can be classified as dynamic and static analysis techniques [RAM75]. Dynamic analysis of a program involves executing the program with test cases and analyzing the output for correctness, while static analysis includes such techniques as program graph analysis and symbolic evaluation [ADR82].

A dynamic test strategy is a method of choosing test data from the functional domain of a program. It is based on criteria that may reflect the functional description of a program, the program's internal structure, or a combination of both [ADR82]. These criteria specify the method of test case generation to be used for a dynamic test strategy. The two dynamic test strategies generally recognized are functional testing and structural testing. These will be detailed in the next subsections.

2.2.1 FUNCTIONAL TESTING

Functional testing involves identifying and then testing all functions of a program (from the lowest to highest levels) with varying combinations of input values to check for correctness of output [BEI84, HOW86]. Correctness of output is determined by comparing the actual output to the expected output computed from the functional specifications of the

program. The internal structure of the program is not analyzed, thus functional testing is often called "black box" testing.

The specifications are used to define the domain of each variable or its set of possible values. Since the program has input and output variables, selection of test data must be based on the input and output domains in such a way that test cases force (or try to force) outputs which lie in all intervals of each output variable's domain. Howden explains the importance of testing endpoint conditions as well as any special mathematical conditions (such as division by zero) that may be encountered in the software [HOW80]. In his approach to functional program testing, Howden also discusses exercising such program elements as array dimensions and subprogram arguments.

Functional program testing has been used as the basis for several combinations of test strategies with reportedly good results [FOS80, HOW80, HOW86, RED83]. These test strategies consist of the test data selection rules of functional testing as well as the test coverage measures found in structural testing techniques.

Random testing is another form of "black box" testing, since the internal structure of the program is not considered when developing test cases. While this method is generally viewed as the worst type of program testing, it does provide "... very high segment and branch coverage" [DUR84]. When combined with extreme and special value testing, it can be an effective method while providing a direction for the generation of further test cases [VOU86].

2.2.2 STRUCTURAL TESTING

Structural testing uses the internal control structure of a program to guide in the selection of test data [BEI84], and it is sometimes known as metric-based test data generation. Coverage metrics are concerned with the number of a program's structural units exercised by test data. Test strategies based on coverage metrics examine the number of statements, branches, or paths in the program exercised by test data. This information can be used to evaluate test results as well as generate test data [ADR82]. Howden and others have discussed path and branch testing strategies [G0075, HOW76, HOW78a], while other strategies such as the use of data flow analysis for obtaining structural information have been proposed and studied [LAS83]. Symbolic evaluation, while considered to be either static or dynamic analysis, is similar to structural testing. This will be discussed in a later section.

A program's control can easily be represented as a directed graph [BEI84, RAM66, SHO83] from which program paths may be identified. It can be shown that for many programs (especially programs with loops) the number of possible paths is virtually infinite [BEI84, HOW78a, WOO80], thus leading to the problem of determining which paths to choose for testing. Criteria for selecting test paths have been discussed [BEI84, HOW78a, RAM76, SHO83] and include statement, decision, condition, decision- condition, and multiple condition coverage. "Coverage" is said to be achieved if a set of paths executed during program testing meets a given criteria [BEI84]. The problem of finding a minimal set of paths to achieve a particular coverage is discussed by So [VIC84] and by Ntafos [NTA79]. Beizer states that the idea behind path testing is to find a good set of paths providing coverage, prove that they are correct and then assume that the remaining untested paths are probably correct [BEI84].

Once a set of paths providing coverage has been selected, the next step involves generating test data that will cause each of the selected paths to be executed. Methods for generating test data from paths are discussed in [ADR82, HOW76, HOW75, HUA75, RAM76] and others, and center around the idea of solving path predicates (discussed later) or at least determining path data constraints to be used for generating test case data.

2.2.3 NEED FOR BOTH FUNCTIONAL AND STRUCTURAL TESTING

The effectiveness of path testing has been questioned [G0075, NTA84], and studies have shown that the class of errors found by this type of testing is not sufficient for complete testing [G0075, HOW76]. As discussed in [NTA84], "... the main shortcoming of structural testing is that tests are generated using possible incorrect code, and thus, certain types of errors, especially errors in the specifications, are hard to detect."

Indeed, Rubey notes that "... there is no single reason for unreliable software", and then he states that "... no single validation tool or technique is likely to detect all types of errors" [RUB75]. He also points out that even though a program fulfills its specifications, it could have specification errors which would render the program unreliable. Glass draws similar conclusions when discussing testing methods [GLA81]. Therefore, since no one testing approach is going to solve all testing problems, functional and structural testing techniques should be considered complementary methods [HOW80].

2.2.4 OTHER TEST STRATEGIES

2.2.4.1 MUTATION TESTING

Mutation testing is considered to be a new error-based testing method [ADR82, VIC84] that is capable of determining the number and kinds of errors that a test data set is capable of uncovering [DEM78]. Mutation testing is based upon two assumptions: 1) the program being tested is nearly correct, and 2) test sets that uncover single errors will also be effective in uncovering multiple errors [ADR82]. The later assumption is known as the coupling effect hypothesis and is described by DeMillo in [DEM78]. He states that "...complex errors are coupled to simple errors" and the effect can be observed in real test/debug situations. Therefore, when testing, attempts should be made to systematically uncover simple errors that may (or may not) eventually lead to complex errors.

Mutation testing involves creating a number of program mutations, with each of the mutations containing different simple errors. For each set of test data there are only two possible outcomes after execution: 1) a mutation gives different results than the original program, or 2) the results are the same. If different results are obtained from the mutation, then the test data were capable of discovering the seeded error in the mutation. Otherwise, one of the following two conditions is true: 1) the test data were not adequate for uncovering the error, or 2) the mutation is equivalent to the original program. Assuming that the second condition is not true, it would be necessary to find more sensitive test data to discover the seeded error. When test data fail to find the seeded error, the programmer should also examine the code to determine the reason. If all errors are discovered by the test data and an adequate number (as defined prior to analysis) and variety of mutations was used, then it can be assumed that the test data set was adequate [DEM78].

Howden has proposed a "weaker" mutation testing technique that is more effective than branch coverage, but less costly and less effective than mutation testing [HOW82b]. In his technique, Howden considers five elementary program components to be used in the mutation process: 1) variable references, 2) variable assignments, 3) arithmetic expressions, 4) relational expressions, and 5) Boolean expressions. One of the main differences and advantages of this technique is that weak mutation testing does not require a separate program execution for each mutation, thus reducing testing time. Weak mutation testing does have the disadvantage of not being able to "... guarantee the exposure of all errors in the class of errors associated with the mutation transformations."

2.2.4.2 DOMAIN TESTING

Domain testing is a strategy designed to detect errors in the control flow of a program (called domain errors), and it is considered to be fairly new and experimental [VIC84, WHI80, WHI86]. The strategy generates test data to examine the input space domain of a program, which is defined as a set of input data satisfying a path condition. In describing the strategy, White and Cohen state: "the control flow statements in a computer program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path" [WHI80]. The strategy is based on the geometric analysis of a domain boundary. A boundary represents the range of input values that will drive the predicate for a given path. Each boundary consists of border segments, which are determined by the conditions of a path predicate. By generating test points on or near the domain borders (since these test points are most sensitive to domain errors), it is possible to detect whether a domain error has occurred [TAI80, WHI80]. An analysis of input space subdomains is discussed in [WEY80] as an extension of the theories of testing proposed by Goodenough and Gerhart in [GOO75]. Domain errors are further defined in the Software Errors section below.

2.2.4.3 SYMBOLIC EVALUATION

Symbolic evaluation is generally considered to be a static analysis technique for testing software [ADR82, VIC84] and involves building and solving (if possible) path predicates to generate test data. Unsolvable predicates indicate infeasible paths in the software which usually raises an error condition [CLA76]. The test data may be used to actually execute the software; thus, symbolic evaluation is an effective way of generating test data for structural testing techniques [GOO75]. This idea is the basis for generating test data in the QUEST automated software testing system and others [BRO86a, CLA76, HOW78b].

Each decision node along a given path will add a term to the path predicate. Further, any of the variables within these terms that are modified by assignment statements must be incorporated into the path predicate such that it can be stated in terms of the input variables. Backward substitution has an advantage over forward substitution in that no space is required for storing the intermediate symbolic values of variables [RAM76]. The process of traversing the path and building the path predicate according to each statement along the path is called "dragging" the path predicate along the path [HUA75]. There is a partial predicate associated with each control statement along the path called a branch predicate. As each branch predicate is added to the path predicate, a new constraint is placed on the values that the input variables may have [CLA76]. Each new constraint should be checked for consistency with the path predicate as it is being built. If an inconsistency is found, the path can be labeled as infeasible [CLA76]. Forward substitution has the advantage of allow-

ing "...early detection of infeasible paths with contradicting input constraints" [RAM76]. Otherwise, the predicate, which must be satisfied by the input data to drive a given path, is stated purely in terms of the input variables.

2.3 AUTOMATION

There are many facets of the testing process which are ripe for automation. As expressed above, the purpose of automation is to enable more and better test cases to be executed in order to provide more reliable code within the testing resource constraints. Classical tools include test harness and instrumentation. More recent literature suggests the need for automating test case generation, regression testing, and even the oracle. These are discussed in the following subsections.

2.3.1 THE NEED FOR AN ORACLE

An oracle is defined to be an external source of information used to detect occurrences of errors. Oracles may be detailed requirement and design specifications, examples, or simply human knowledge of how a program should behave. Theoretically, an oracle is capable of determining whether or not a program has executed correctly on a given test case [HOW86]. Practically speaking, the manual effort needed to verify test results makes this the most labor-intensive part of the testing process [BRO87].

Some type of oracle must be employed, either by test personnel or by an automated testing system, to determine whether outputs are correct. Two types of oracles that could be integrated into an automated testing environment are design specification simulators and redundant coding. A paradigm for integrating such an automated oracle into the testing process was given by Brown [BRO87].

2.3.2 AUTOMATED TESTING TOOLS

2.3.2.1 STRUCTURAL TESTING TOOLS

A path predicate states a set of conditions that must be satisfied in order for a path to be traversed. As each branch is added to the path predicate, a new constraint is placed on the values that the input variables may have [CLA76]. Thus the predicate, which must be satisfied by the input data to drive a given path, is stated purely in terms of the input variables.

A predicate may be simplified and then translated into a series of inequalities for solution, thus generating test cases. Linear inequalities can easily be solved if variable data types are limited to integer and real, while non-linear cases are much more difficult and require other less formal methods which use the generated constraints [CLA76, HOW75, RAM76].

Other problems affecting the solution of linear predicates include: 1) array subscript variables which are dependent upon input data, 2) loop structures, 3) subprogram interfaces, and 4) global variables [CLA76, HOW75, RAM76]. Another approach to testing closely related to predicate solution is that of symbolic evaluation. Several automated systems for performing symbolic evaluation exist [CLA76, HOW78b].

2.3.2.2 FUNCTIONAL TESTING TOOLS

The goal of functional testing is to design and execute a set of test cases that exercise the entire functionality of the software [OST86]. Numerous methods have been described for selecting specification-based test data [MYE79, WEY80, HOW81, OST79]. Also, tools have been developed to assist in the generation and maintenance of specification-based test cases [OST86, SOL85, CER81, CHO86, BOU85]. However, these tools require considerable user interaction, and they do not fully automate the process of test data generation.

Tools have been developed for static analysis, dynamic testing, and the facilitation of regression testing [TSA86]. The extension of these tools to include concurrency constructs is in its infancy [GOR86]. Concurrency has been studied in terms of structural testing [TAY86], as well as static analysis with symbolic execution [YOU86]. The use of symbolic execution has been extended to a tasking subset of Ada [DIL86], to explore "safety properties", such as mutual exclusion and freedom from deadlock.

2.4 RELIABILITY MODELS AND TEST ADEQUACY CRITERIA

Attempts have been made to quantify the reliability of software entities being tested. Statistical models for various testing approaches have been derived and applied [DUR80, ROS85A, DUR81, ROS85B]. As in all applications of statistical modeling, assumptions and approximations must be made. Although such models are not generally accepted as perfect indicators of software reliability, coverage metrics will continue to be used as indicators of software reliability until this area has advanced far beyond its present state.

Since the purpose of testing is to determine whether a particular piece of software contains faults, an ideal test set would succeed only if the software contains no faults [GOO75]. Unfortunately, it is not generally possible to derive such a test set for a program, or to know that a test set is ideal. We must use some test adequacy criterion to determine how close our test set is to ideal and when to stop testing. Such a criterion is called program-based if it is independent of the specification of the program, and so is based purely on the code. Statement coverage and branch coverage are two program-based test adequacy criteria [WEY86].

Instrumentation of programs aids in evaluating the degree to which an adequacy criteria have been met. Instrumentation is the insertion of additional statements into the program which, when the program is executed, will compute some dynamic attributes of the program [HUA78]. For instance, a simple instrumentation scheme would insert counters to record the number of times each statement is executed. Instrumentation to compute certain program-based adequacy metrics allows the testers to evaluate their progress.

The adequacy measures produced by instrumentation may be classified as control-flow coverage measures, data-flow coverage measures [FRA88], and most recently data coverage measures [SNE86]. One data-flow coverage measure is definition-reference chain (dr-chain) coverage, which is concerned with the definition and referencing of program variables [HOW87, WIL85, RAP85]. Statement and branch coverages are examples of control-flow coverage measures. Recent work has been performed in developing adequacy criteria derived from data flow testing criteria [FRA86], and in comparing the various criteria [CLA86]. Some experimental comparisons suggest that the various approaches should be considered as complementary rather than competing [GIR86].

2.5 TEST DATA GENERATION

A software testing problem that is very closely related to test set evaluation is that of test data generation. Quite often, the difference between the two blurs because test data generation schemes generally attempt to generate data that will satisfy some specific test data adequacy criterion. Test data generation has been defined as consisting "of specifying and providing the test input data and of calculating the test output data" [VOG85].

Generating test inputs for a program may not appear to be a difficult problem since it may be done by a random number generator [DUR81]. However, although random testing alone has been shown to be an inadequate method for exposing errors, when combined with extremal and special value (ESV) testing, it can be an effective method and can provide a direction for the generation of future test cases [VOU88]. On the other hand, algorithms for generating test data to satisfy particular adequacy criteria have generally had very bad time and space complexities and produced small amounts of test data. In fact, it is not possible (i.e., there exists no algorithm) to generate test data which causes the execution of any arbitrary program path [MIL84].

DeMillo, Lipton, and Sayward [DEM78] attempted to develop a practical test data generation methodology somewhere between random data generation and full program predicate solution. Noting that programmers produce code that is very close to being correct, they observed a program property which they named the coupling effect. Basically, the coupling effect is the ability of test cases, designed to detect simple errors, to surface more subtle errors as well. Howden, on the other hand, developed a set of functional testing rules [HOW87]. Although both of these research efforts were directed at helping programmers test their code, they are also directly applicable to automatic test data generation. They are not algorithms, but instead are useful rules of thumb. Such rules are typically referred to as heuristics, which embody certain bits of "expert knowledge." Thus, a knowledge-based or expert system approach is very appropriate in attacking the problem of generating test data for software programs. This approach is made possible not only by the maturing body of knowledge about software testing, but also by developments in the field of rule-based systems, a branch of artificial intelligence.

2.6 THE PATH/PREDICATE SOLUTION PROBLEM

As stated earlier, test data generation algorithms are usually designed to generate test data sets which satisfy some particular test adequacy criterion. Since algorithms such as these are provably nonexistent for a general program, the domains of the algorithms are some subset of all possible programs. One such subset is the set of all programs with only linear path predicates. The applicability of each technique is, of course, limited by its restricted domain. This limitation is the first problem with conventional test data generation algorithms. The second problem with such algorithms is that they usually have very bad time and space complexities. For example, the path-predicate generation/solution approach for statement coverage must: (1) choose, from the (possibly infinite) set of possible paths through the program, a subset of these paths which will provide statement coverage, (2) construct a path predicate for each chosen path, and then (3) solve the associated path predicate for each path in terms of the inputs to the program. The predicate solution problem alone is very complex, and no algorithm exists for solving general nonlinear predicates [MIL84]. However, there are some good methods which will find solutions to many predicates.

One implementation of the path predicate methodology is the QUEST testing tool [BRO86, WEY88]. QUEST is applicable to a subset of FORTRAN 77 and provides path predicate generation options which attempt to generate test data to satisfy the statement coverage, decision coverage, condition coverage, or decision/condition coverage test adequacy criteria. Of course, there is no guarantee that the predicate solution algorithm will be able to solve a given predicate; it must halt after a predefined number of unsuccessful attempts to find a solution. Even with the ability to solve predicates, each solution yields input data for only one test execution. This is the third problem with traditional test generation methods - they produce a relatively small number of test cases.

2.7 CONCLUSION

While QUEST/Fortran aided the testing process by automating some structural testing techniques, its use of symbolic evaluation leads to a number of problems: 1) limitations on the program structure which could be handled, 2) poor space-time efficiency of solving a predicate for each program path, 3) the limited number of test cases that could be generated in a given amount of time, 4) the limitations of the algorithms used to solve the path predicates, which sometimes meant that obvious path predicates were labeled as unsolvable and 5) the generation of trivial test cases.

QUEST/Ada addresses the problems encountered with path predicates by generating test cases using a rule base as opposed to symbolic evaluation. While the traditional instrumentation techniques will be used to evaluate coverage, unlike QUEST/FORTRAN, QUEST/Ada uses a formal parser/scanner to enable the instrumentation capabilities to be easily generalized. Further, the information obtained from this instrumentation upon execution will be fed back to the test data generator to successively improve the quality of the test cases. These innovations make QUEST/Ada a unique approach to software testing.

3. DEFINITION OF SYSTEM STRUCTURE

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 Appendix A, some of these diagrams will be used in this section for illustration. Figure 1 shows the highest level of data flow, with the user interacting with the test environment, called QUEST (Query Utility Environment for Software Testing). 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_ADA-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_ADA-21.

Figure 2 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 report generator (RGEN). The interfaces between the various subsystems are listed in Table 1 and described in the following section.

Table 1. Description of High Level Interfaces

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

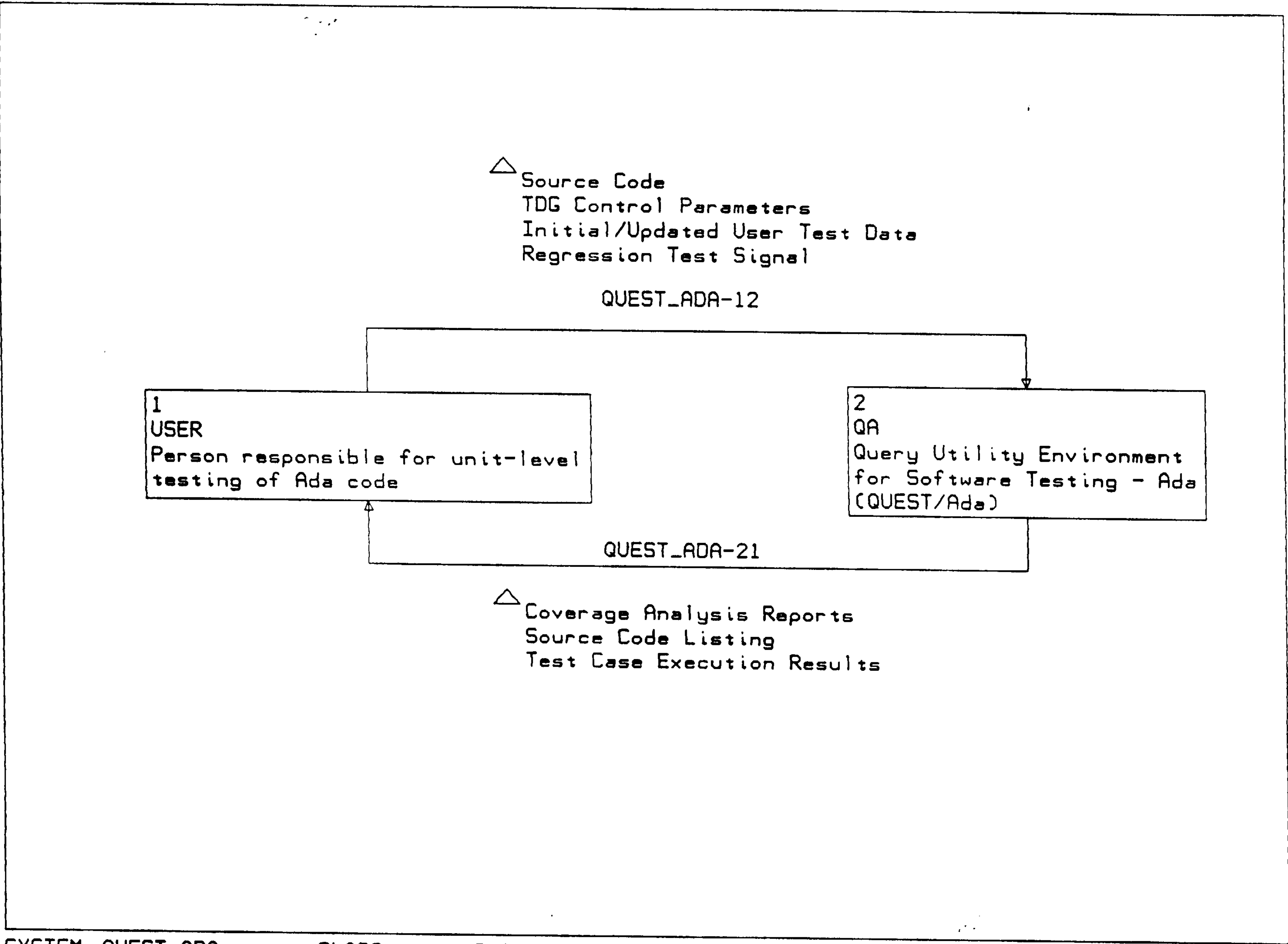


Figure 1 - Top Level IORL Description

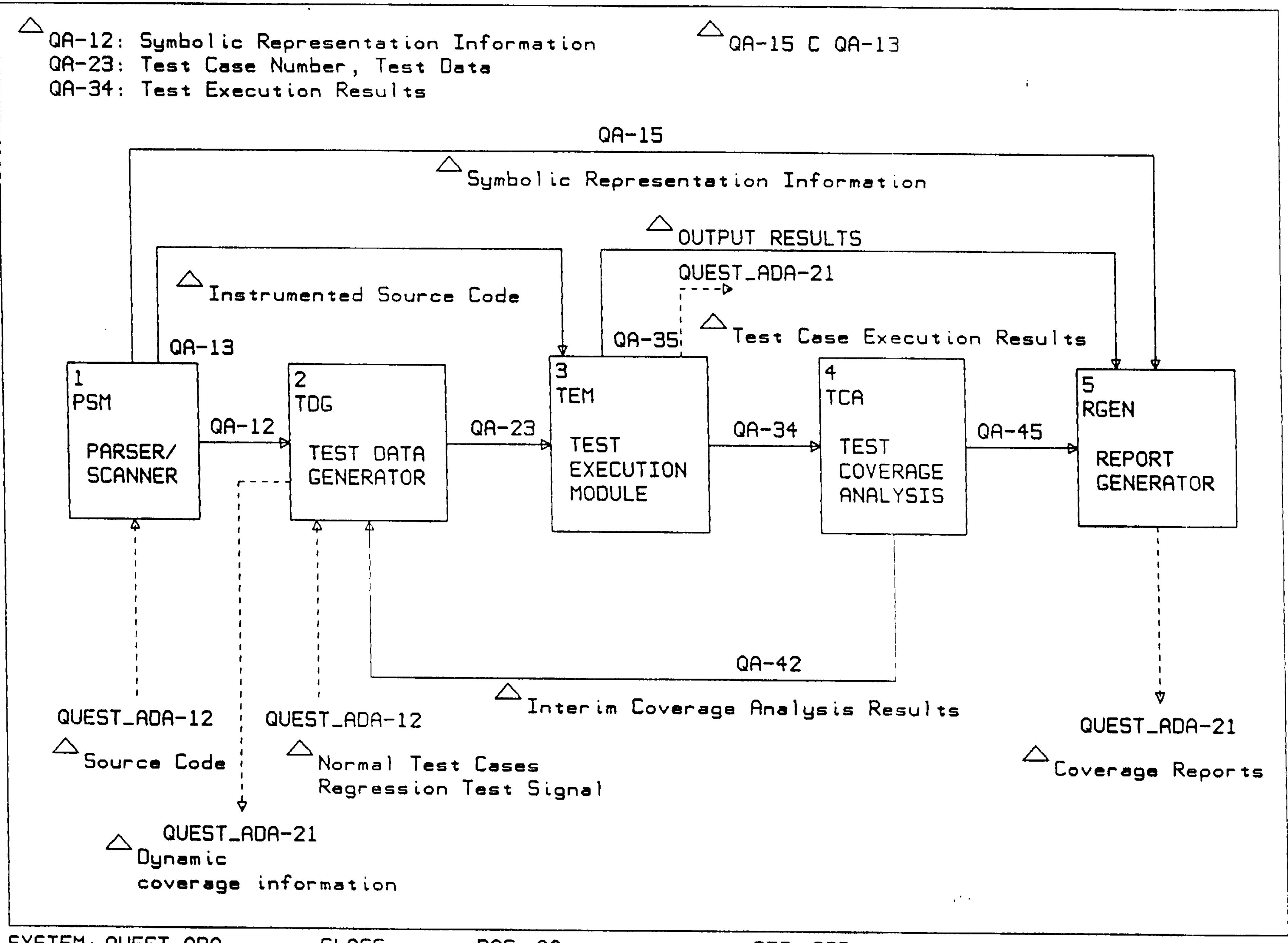


Figure 2 - Detailed IORL Description

4. DEFINITION OF HIGH LEVEL INTERFACES

4.1 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 2.

Table 2. PARSER/SCANNER MODULE INTERFACES

```
INPUT:      QUEST_ADA-12, 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: REPORT GENERATOR
                    1. DECISION/CONDITION LIST
                        a. DECISION NUMBER
                        b. CONSTRUCT TYPE
                        c. NUMBER OF CONDITIONS
```

4.2 TEST DATA GENERATOR INTERFACES

The Test Data Generator (TDG) interfaces are given in Table 3. 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_ADA-12) to generate additional test data. User-defined test data may also be supplied at any of these check points.

```
-----------------------------------------------------------------
              Table 3.  TEST DATA GENERATOR INTERFACES
-----------------------------------------------------------------

INPUTS:      QUEST_ADA_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_ADA-21, DYNAMIC COVERAGE INFORMATION
                TO: USER

-----------------------------------------------------------------
```

4.3 TEST EXECUTION MODULE INTERFACES

The Test Execution Module (TEM) interfaces are shown in Table 4. 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 Report Generator.

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. The specific format of the coverage library will evolve as the reporting requirements evolve.

```
-----------------------------------------------------------------
              Table 4.  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: REPORT GENERATOR

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

4.4 TEST COVERAGE ANALYSIS INTERFACES

Table 5 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. 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: REPORT GENERATOR
                    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
```

4.5 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 6.

Table 6. Report Generator 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:     QUEST_ADA-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. SCOPE OF THE PROTOTYPE

The formidable task of constructing a working prototype of an automated testing environment during a one-year period required a clear definition of the scope of the project. Since the goal of the prototype was to demonstrate the feasibility of automatically generating test data for a variety of Ada modules, these limitations will be based on the data types allowed as input to the modules being tested.

The initial prototype generates test cases for multitasking Ada programs. Standard coverage metrics will be calculated for these programs. However, they will not necessarily be an effective indication of program correctness, due to the unpredictable nature of rendezvous sequences. Consideration has been given during the prototype design and development to establish approaches for handling concurrency. However, the actual prototyping of these approaches has been deferred until Phase 2.

The prototype produced in phase one of the QUEST/Ada project represents a complete basic automatic rule-based testing environment. In addition to the rule base, the prototype includes the ability to collect relevant testing information from an Ada module and a testing coverage analyzer. The development of this complete environment provides the capability for evolving a sophisticated set of testing heuristics by using statistical methods to evaluate large numbers of test cases across a wide cross section of programs. Even the most facile large-scale quantitative evaluation techniques are enormously complicated to perform by hand, and more complicated procedures are virtually impossible to do manually in any reasonable amount of time. Using the QUEST/Ada prototype, however, the speed at which large-scale evaluations can be performed is limited only by the speed of the associated hardware.

Notably absent from the prototype is the ability to automatically "instrument" the Ada code to be tested (see the Parser/Scanner section for an explanation of instrumentation). This capability would allow code to be placed automatically into the testing environment. While this ability would be important to a production version of QUEST, it has become clear from the prototype that such an automatic instrumentation tool would be even more complicated than a full Ada parser. Consequently, while the tasks involved in producing the automatic instrumentation tool are quite straightforward, the development would nonetheless require several times the manpower available to this project. Thus, while the prototype proved the feasibility of this approach, subsequent tests of the remaining components of the QUEST system will rely upon manually-generated instrumentation.

The main effort in developing the prototype parser concentrated on extracting information critical to the automatic testing process, such as variable types, bounds and names. These are essential to the automatic generation of test cases and the coverage evaluation. Hand instrumentation, which was found to be a fairly simple task, was used to prepare code for the testing environment.

Finally, it is notable that the subset of Ada covered by the initial prototype is small. As the emphasis of the first phase of the QUEST project lay in establishing the validity of the rule-based automatic testing paradigm and providing an environment for the efficient quantitative evaluation of testing heuristics and techniques, the development effort was directed toward a fully functional small subset prototype rather than a skeleton prototype of a more robust subset. This approach additionally allows for a more intelligent expansion to a wider set of test heuristics, since the tools for evaluating the quality of new heuristics are now complete. The broadening of the Ada subset is one of the primary tasks of Phase two.

6. PARSER/SCANNER DESIGN

6.1 GENERAL PARSER SCANNER REQUIREMENTS

The parser/scanner module is responsible for building the data structures required by the rest of the QUEST system and creating a listing of the source code for use by the tester. Information contained in these data structures must identify the control constructs, global variables referenced (i.e., altered) within the module, and parameters input to the module.

As mentioned above, the roles of instrumenting the Ada source code and surrounding the module under test with an execution driver or test harness is to be performed manually. Instrumentation of the Ada source code is required for determining test coverage and for providing feedback data required by the AI test data generator. Each decision and condition in the program must be instrumented so that all of the standard coverage metrics may be calculated by the report generator. The feedback data are used as an indication of test case quality for directing the generation of new test data. While this function is not a part of the prototype, its integration into the Ada compiler in a production test system is straightforward.

The data structures built by the prototype parser provide information concerning the structure of the module under test. This includes information about the number and types of input variables and parameters, the statements and segments executed as a result of decision outcomes, and the structure of decisions and conditions. These data structures are used by the test data generator and the report generator modules.

A listing of the source code is provided to the tester as an aid in analyzing the output of the report generator. As an option to the user, this listing shows the embedded instrumentation code. Unique identification numbers are assigned to each decision, condition, and code segment in the original code listing.

Two parser/scanner generator packages, LALR 3.0 and BISON, were evaluated for use in producing an attributed grammar to provide the parser/scanner functions. These were selected because of their advertised capabilities to handle the large number of productions required by the Ada grammar. While LALR 3.0 appeared to function on some small examples, there was no evidence that it could handle the complete Ada grammar. On the other hand, BISON has shown great promise as illustrated by the example presented below.

6.2 PARSER/SCANNER OVERVIEW

The parser scanner first determines, by a syntax-directed parse of the Ada source code, the appropriate information for the expert system's test case generator. It builds a symbol table containing this information and then writes information from the symbol table into the knowledge base of the expert system. Currently the instrumented code is surrounded by a manually-generated Ada source driver so that the unit under test can be compiled by the host Ada compiler. Machine independence is achieved by having the test module be a high level unit and not dependent on any particular compiler or its symbol table representation.

In order to maintain flexibility and allow the prototype's parser/scanner to be extended to other forms of testing, the parser scanner unit is generated by a parser generator tool and a LALR description of Ada. As with the expert system, where rules may be

inserted or deleted without major modification of the system itself, the actions of the parser generator may be modified or extended to reach a goal of a full Ada testing system.

6.3 LIMITATIONS OF THE PARSER/SCANNER PROTOTYPE

Because the primary goal of the first phase of the project was a working prototype of the testing environment, only a subset of the Ada language was handled by the system. This required decisions as to which capabilities should be included in the prototype and which could be safely omitted. Nevertheless a complete LALR description of Ada is used to avoid theoretical limitations.

Certain tradeoffs were made in the decision to use a high level test module in order to avoid host or compiler dependencies. The most significant sacrifice is that all interfaces between the unit under test and the prototype are a part of the program when it is executed. Limitations, such as the size constraint of the users' systems become limitations of the prototype. More seriously, errors in the user's module remain as errors of the test module.

Errors in instrumentation or the harness/driver for the module could result in false error detection for the module under test. The remedy for this, which is strongly advised in any event, is to run the uninstrumented module in parallel with its corresponding instrumented code to assure that no such errors have been introduced.

Although it avoids compiler dependencies, the decision to use a high-level parser generator tool to generate symbol table information adds an overhead to the resulting system. As presently implemented, each program unit which is to be visible to the prototype must be included with the unit. Extending the prototype to include Ada packages will require a more sophisticated data structure for the symbol table. If extensions are made to include separate compilations, or packages with code not visible to the module under test, some form of library management interaction with the host system and possible sharing of it's compiler symbol table information might be required. However, this is not seen to be as much of a problem with unit-level testing as it will be when QUEST is used for integration testing.

The required instrumentation routines have been written to support the condition branch coverage tested by the expert systems. The expressions handled by the prototype are relational operations of arithmetic expressions, and Boolean operators are not allowed. The reason for this restriction is that meaningful heuristics have been found in previous work for path expression coverage of conditions determined by the arithmetic and relational operators. Restricting use to these heuristics would enable more meaningful testing of the prototype.

Control flow determined by values of access types is not handled by the prototype. Consideration of these types, and heuristics for testing code containing conditions determined by the values of these types, is of interest in future development, since many programmers have considerable difficulty when dealing with these types. Similarly, information-altering control flow that is determined by external sources, such as I/O or tasking entry, have been excluded from the initial prototype. Future work with the prototype will heuristically address these issues.

In summary, the major work done on prototyping the parser/scanner was to develop the code to: (1) generate the symbol table to hold type and subtype information required for

meaningful test case generation, (2) generate the routines to write this information as needed by the expert system, and (3) generate the Ada routines called by the instrumented Ada code for input and output of the values of the instrumented boolean conditions. So far, heuristic rules utilizing routines of the prototype have been used with integer based types, subtypes, and user- defined types based on this type. The parser/scanner symbol table routines are written to gather the information required to support other numeric and enumeration types as well as those user-defined types based on these types.

6.4 EXAMPLE MODULE INSTRUMENTATION

Listing 1 provides an example of uninstrumented code that has been tested by the system. The data structure used to implement the symbol table is a linked list of linked lists, as pictured in Figure 3. Listing 2 shows a sample from the LALR description of Ada used by the prototype's parser scanner, and the semantic actions used to generate the symbol table. Listing 3 shows the instrumented code and the surrounding module for the previous uninstrumented Ada code example. Finally, Listing 4 presents a fact asserted by the parser scanner into the knowledge base of the expert system.

Un-instrumented Ada Code

```
procedure test(i: in out integer;
               j: in out integer;
               k: in out integer) is

begin
    while i > j loop
        i := i - 1;
        k := (k + 314) mod 25;
        if  i > k then
            while i > k loop
                k := k + 1;
                if  k >= 27  then
                    null;
                else
                    null;
                end if;
            end loop;
        else
            if  i < k-3 then
                if  i-10 < j then
                    null;
                else
                    null;
                end if;
            else
                 while  i >= k-3 loop
                    i := i - 1;
                end loop;
            end if;
        end if;
    end loop;

    if i = j then
        null;
    else
        null;
    end if;

end test;
```

Listing 1 - Uninstrumented Code

Symbol Table

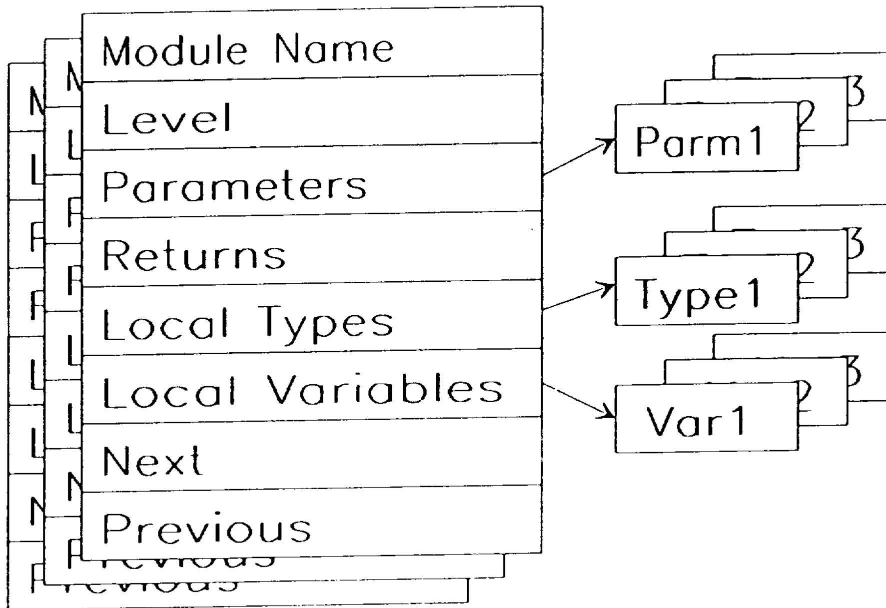


Figure 3 - Parser-Scanner Date Sturcture

Ada Grammar with Semantic Actions

```
subprg_body : subprg_spec IS_
              { $<node>1->level = level++; }
              .decl_part.
              BEGIN_
                  seq_of_stmts
              .EXCEPTION__excptn_handler..excptn_handler...
              END_ .designator. ';'
              {
                level--;
                $<node>1->last_line = lines;
                if (!strcmp($<node>1->name, module_under_test))
                      YYACCEPT;
              }
            ;

subprg_spec : PROCEDURE_ identifier
              {
                $<node>$ = make_mod();
                $<node>$->name = $<str>2;
                $<node>$->first_line = lines;
              }
              .fml_part.
              {
                $<node>$->parms = $<parm_l>4;
                $<node>$->returns = NULL;
              }

fml_part    : '(' prm_spec .._.prm_spec.. ')'
               { $<parm_l>$ = link_parms($<parm_l>2,$<parm_l>3);}
            ;

prm_spec    :  idents ':' mode ty_mk ._ASN_expr.
               { $<parm_l>$ = make_parms($<id_l>1,
                                          $<val>3,
                                          lookup_type($<str>4)); }
            ;

idents      : identifier ...ident..
              { $<id_l>$ = make_id($<str>1,$<id_l>2); }
            ;
```

Listing 2 - Sample LALR Description

Instrumented Ada Code

```
with text_io, instrumentation; use text_io, instrumentation;

procedure driver is
     BeginTest:     boolean;
     TestNum:       integer;
     indata,
     outdata:       file_type;
     i,j,k:         integer;

     package inst1 is new instrumentation.integer_inst(integer);
     use inst1;
     package int_io is new text_io.integer_io(integer);
     use int_io;

procedure test(i: in out integer;
               j: in out integer;
               k: in out integer) is
begin                                                               --
    BeginTest := FALSE;
    while  relop(TestNum,1,0,i,GT,j)  loop                          --d1
        i := i - 1;
        k := (k + 314) mod 25;
        if  relop(TestNum,2,0,i,GT,k)  then                         --d2
            while  relop(TestNum,3,0,i,GT,k)  loop                  --d3
                k := k + 1;
                if  relop(TestNum,4,0,k,GE,27)  then                --d4
                    null;
                else
                    null;
                end if;
            end loop;
        else
            if  relop(TestNum,5,0,i,LT,k-3)  then                   --d5
                if  relop(TestNum,6,0,i-10,LT,j)  then              --d6
                    null;
                else
                    null;
                end if;
            else
                 while  relop(TestNum,7,0,i,GE,k-3)  loop --d7
                    i := i - 1;
                end loop;
            end if;
        end if;
    end loop;

    if relop(TestNum,8,0,i,EQ,j) then                               --d8
        null;
    else
        null;
    end if;
end test;
```

Listing 3 - Instrumented Code

```
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
        BeginTest := TRUE;
        get(indata,TestNum);                 -- get test data
        get(indata,i);
        get(indata,j);
        get(indata,k);

        test(i,j,k);                         -- run 1 iteration

        put(outdata,TestNum);                -- put OUT parameters
        put(outdata,i);
        put(outdata,j);
        put(outdata,k);
        new_line(outdata);
    end loop;

    close(indata);
    close(intermediate);
    close(outdata);
end driver;
```

Listing 3 - Instrumented Code (Continued)

Rules Asserted into Clips Fact Base

```
(deffacts parser_scanner_assertions ""
     (names i j k)
     (types int int int)
     (low_bounds 1 2 3)
     (high_bounds 100 50 25)
)
```

Listing 4 - CLIPS Assertion Rule Example

6.5 PARSER/SCANNER IMPLEMENTATION DETAILS

The Parser/Scanner Module, as currently implemented, operates in two passes. The first pass, called SYS_SCAN, creates a doubly linked list of module names to be returned to the user interface. The only semantic actions required in this pass are those for creating a node, filling in the module name, and appending the node to the linked list. The second pass, SYS_PARSE, is considerably more complex. During this pass a symbol table consisting of multiply linked lists is created which contains all the information necessary for test data generation and report generation. In the production version of QUEST, this module would also be responsible for automatic instrumentation. Thus, ultimately consideration should be given to integrating this portion of QUEST into the Ada compiler.

Upon completion of SYS_PARSE, a subroutine, ASSERT_FACTS(), traverses the symbol table gathering and writing information to a fact base used by the Test Data Generator. Information required by the Report Generator is currently stored in global variables. The majority of the semantic actions in SYS_PARSE create and link various node types into the symbol table. Many of these routines are quite similar in function -- the main difference being the type of node upon which they operate. These routines will be described individually below.

The full symbol table is maintained during SYS_PARSE. This is somewhat different from the usual method of maintaining a stack of (partial) symbol tables, one for each module. The reason for maintaining the full symbol table in memory is to allow information to be passed to the other parts of QUEST at the end of the parse. It would also serve in the future to facilitate the instrumentation parse by saving the scoping and variable usage information.

Scoping information is currently maintained as an integer 'level' associated with each module. The lowest level corresponds to the outermost module. Listing 5 gives an example which demonstrates the level concept. A module has access to it's surrounding module (the previous module in the list with a level one less than it's level). The surrounding module has access to it's surrounding module, etc. This method of scoping can be further extended to include Ada packages.

Listing 5. Example to Illustrate QUEST Parser Levels

```
LEVEL     MODULE
-----     ------------------------------------------------
0           procedure outermost;
1                 procedure inner1;
                        {inner1 body}
                    end inner1;
1                   procedure inner2;
2                       procedure innermost;
                              {innermost body}
                        end innermost;
                        {inner2 body}
                  end inner2;
            end outermost;
```

Listing 6 presents the types which are declared for use as semantic values during SYS_PARSE.

Listing 6. Types Declared During SYS_PARSE

```
int                         val;
   integer values

  char                                *str;
  character strings  -  identifiers for modules,
  types, variables,...

  struct rng_decl              *range;
  rng_decl - subtype range declaration.

struct str_list                      *id_l;
  id_l - linked list of identifiers. Used for
  temporary storage of <idents>

  struct var_list                     *parm_l;
  parm_l - linked list of a module's formal
  parameters.      also includes type info, IN OUT
  info, etc.

  struct var_list              *var_l;
  var_l - linked list of a module's local
  variables

  struct type_list                    *type_l;
  type_l - linked list of a module's local types

  struct module_list                 *node;
  module_list - linked list of modules containing
  scope information (level), formal parameters,
  local types and variables, etc.
```

The following modules are called during SYSPARSE:

(1) make_mod() - creates a module_list node and appends it to the module list:
static struct module_list *make_mod()

(2) link_types() - concatenates two type lists:
static struct type_list *link_types(head,tail)
struct type_list *head,*tail;

(3) make_type() - creates a type list nodes and links it to the front of the list:

static struct type_list *make_type()

(4) link_parms() - concatenates two var_lists:
```
static struct var_list *link_parms(head,tail)
struct var_list *head,*tail;
```

(5) make_parms() - converts an id list to a var list. It is used for declarations such as: id1, id2, id3: type, and it creates a var_list node for each identifier:
```
static struct var_list *make_parms(id_l,mode,type)
struct str_list      *id_l;
int                  mode;
struct type_list     *type;
```

(6) make_id() - creates an id node and link to front of list:
```
static struct str_list *make_id(str,id_l)
        char *str;
        struct str_list *id_l;
```

(7) link_id() - concatenates two id lists:
```
static struct str_list *link_id(head,tail)
        struct str_list *head,*tail;
```

(8) assert_facts() - traverses the symbol table and writes information to the CLIPS fact base:
```
static assert_facts()
```

(9) lookup_predefined() - looks up a predefined Ada type static struct type_list *lookup_predefined(str):
```
char *str;
```

(10) lookup_local() - looks up a user defined type in a module's symbol table:
```
static struct type_list *lookup_local(str,module)
        char *str; struct module_list *module;
```

(11) lookup_type() - looks up a type in the symbol table; it calls lookup_local() for each "visible" module until the type is found or the symbol table is completely searched; if not found it calls lookup_predefined():
```
static struct type_list *lookup_type(str)
        char *str;
```

6.6 FUTURE PARSER/SCANNER RESEARCH EFFORTS

There are four major areas in which the PSM should be extended in the near future. These are: (1) Extending the allowed data types, (2) Extending the scoping information to include packages, (3) Enhancing the amount and type of information passed to the Test Data Generator, and (4) Implementing the automatic instrumentation pass. The first two areas are rather straight-forward and will not be discussed. However, area 3 is where the greatest improvement in the performance of QUEST could be realized.

Currently the information passed to the Test Data Generator (TDG) consists of a list of the parameters of the module under test, their type, and their range of possible values. This information should be extended as much as possible. In particular, data flow analysis could yield much information that is useful to the TDG. For example, it may be determined that one side of a particular condition can be evaluated statically, or that a variable is not modified during a particular segment of code. This is the type of information that would be most useful to the TDG.

Although implementing the automatic instrumentation pass will be a straight-forward (albeit, extremely labor-intensive) procedure, some explanation is in order. Instrumentation of the various flow-control constructs will entail building a syntax tree for the condition expression and then reconstructing the original expression with function calls to the instrumentation package inserted at the appropriate places. As explained above, hand instrumentation will be performed in order to prevent the development of the automatic instrumentation from delaying the development of the remaider of QUEST.

The harnessing of the module in order to drive the module-under-test (MUT) is also considered to be part of the instrumentation. Two methods have been considered for this. The first method consists of "cutting out" the MUT from the original main program and surrounding it with a driver. This method is simple and should probably be considered for the near-term. However, certain modules cause difficulties when tested by this method.

As an example, consider the program given in Listing 7. If the MUT is procedure b(), then procedures b() and a() must both be present in the driver module in order to test b(). However, b() modifies a global variable with the same name as the parameter to procedure a(). Therefore, one of the variables must be renamed to avoid naming conflicts in the driver module.

Listing 7. Example Program

```
        x,y: integer;
        procedure a()
        begin
                modify x;           -- global variable x
        end a;
        procedure b(x:integer);
        begin
                call a()
                modify x;           -- parameter x
        end b;
begin
        call b(y);
end main;
```

A second method for constructing the driver keeps the program structure intact. A Boolean variable, Test, is defined in the outermost module. Each lower level module tests this variable and if it is true, reads in local data referenced by the MUT, as shown in Listing 8. If this variable is false, it executes it's code normally. This method eliminates the need for renaming any variables and allows mutually recursive modules to be easily tested. However, the method is considerably more complex and it is presently uncertain whether it is truly necessary to drive the MUT by this method.

Listing 8. Modified Example Program

```
procedure driver;
Test: boolean;
      procedure main;
            x,y: integer;
            procedure a()
            begin
                  modify x;     -- global variable x
            end a;
            procedure b(x:in out integer);
            begin
                  Test := false;
                  call a();
                  modify x;     -- parameter x
            end b;
      begin
            if (Test) then
                  read(x,y);
            end if;
            call b(y);
      end main;
begin
      Test = true;
      call main;
end driver;
```

7.0 TEST DATA GENERATOR (TDG) DESIGN

The general goal of the Test Data Generator is to provide a prototype for intelligent test data generation. By combining the previous software coverage analysis techniques and the artificial intelligence knowledge-based approach, more efficient test data generation can be achieved. When more test cases are needed to cover a specific branch, heuristics are used to simplify the problem of finding the condition boundaries from which new test cases can be defined. This approach has been found to obtain levels of coverage that are difficult and inefficient to obtain under random test case generation. As the prototype develops, it will be tested against its prior performance to assure that the rules added are leading to greater efficiency.

The method applied by TDG has the objective to achieve a maximal branch coverage of a software package. The analysis of actual coverage follows the Path Prefix Strategy of Prather and Myers [PRA87]. In the Path Prefix Strategy, a software package is represented by a simplified flow chart. Each condition in the flow chart contains two branches, true and false. The goal of a set of test cases is to maximize the number of covered branches. The coverage of these conditions and branches is recorded in a branch-coverage table. In this table each condition contains two entries, one for the truth branch and one for the false branch. When a branch of a condition is covered (or reached), the branch's entry in the table is marked with an "X". In addition to the marking process, the test cases that cover this branch are also recorded.

When new test cases are to be generated, the branch-coverage table is examined to select a condition that is not yet fully covered, i.e., only one branch is covered. After a condition is selected, an associated test case of this condition is retrieved. Since some earlier case started from the package entry point and reached the condition under consideration, it is already "close" to cover the branch that is not yet covered. The Path Prefix Strategy uses an "inverse" approach to generate a new test case. However, as stated in [PRA87], the inverse problem is still not totally understood.

Our approach to intelligent test data generation includes the following tasks: (1) initial case generation, (2) coverage analysis and "goodness" evaluation, and (3) new case generation. Figure 4 shows the relationships between these major concepts. In this system, a test case is represented as (case number, value-1, value-2, value-3, ..., value-n). "Number" indicates the generation sequence of this case. Value-1, value-2, ..., value-n are the values of each input variable of the package. The design and development of the TDG component prototype proceeded with the following simplifying assumption: a condition contains constants, arithmetic and logic operators, and input variables only. Internal variables, i.e., defined within the tested package, would not appear in a condition. This provided a more explicit relation between the input variables and the condition branches. The system is being developed using CLIPS [CLI87], an expert system development tool which provides various interfaces to communicate with external functions written in other programming languages, e.g., C, Pascal, Fortran.

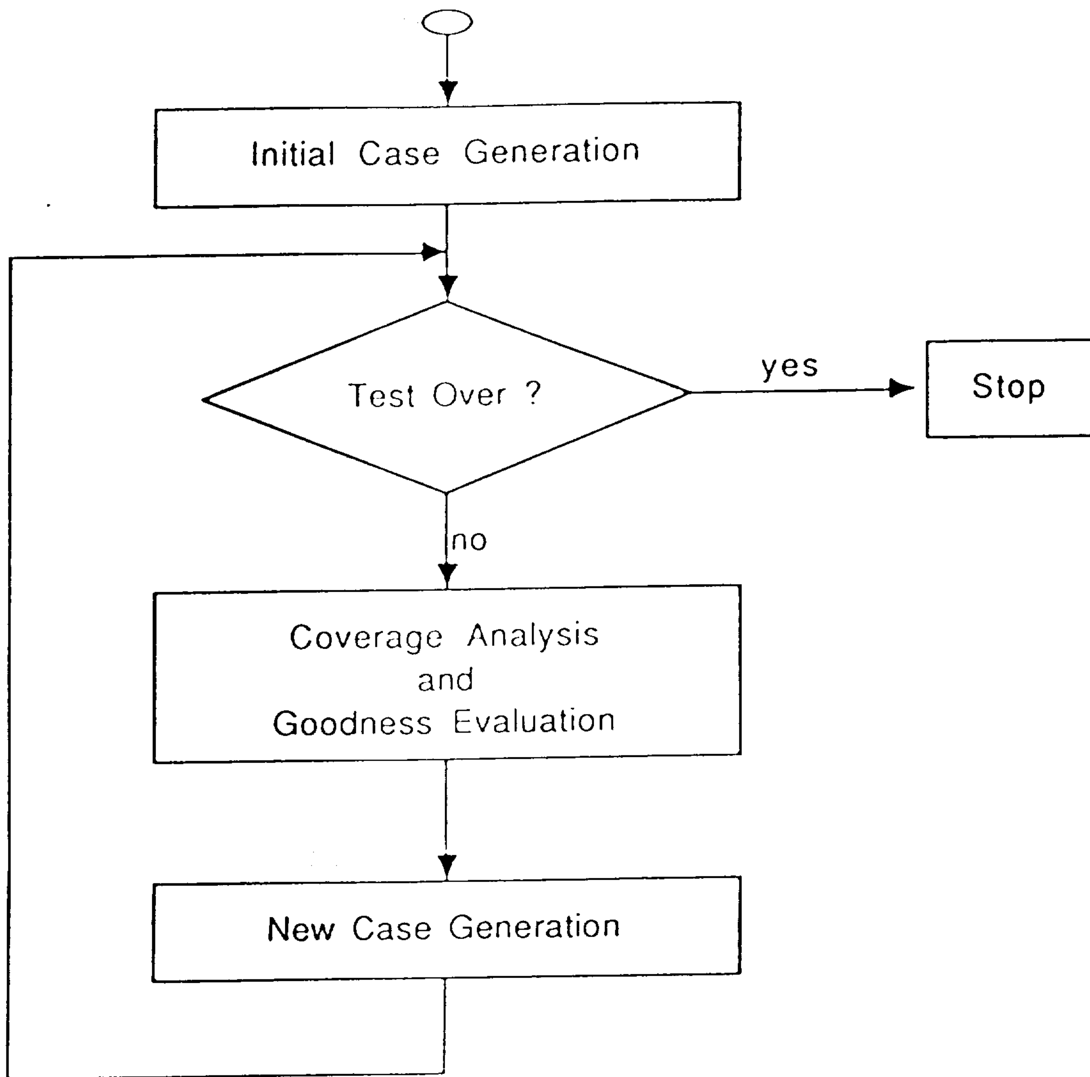


Figure 4 - Flow of Test Data Generator

7.1 INITIAL CASE GENERATION

Initial test cases are needed to start the process. In the event the user provides test cases with the target software package, this initial case generation step can be skipped. However, if test cases are not provided, the initial test cases can be generated based on the syntax information of each input variable, including type, lower limit, and upper limit. (Note: this will be the approach used here in order to provide a common baseline to which improvements in the TDG can be compared.) Four cases are generated: (1) each variable is assigned to its mid-range, (2) each variable is assigned to its lowest value, and (3) each variable is assigned to its highest possible value. For each variable, the mid-range, bottom, and top are defined as:

mid-range = (upper-limit - lower-limit) / 2

bottom = lower-limit + (upper-limit - lower-limit) * 0.05

top = upper-limit - (upper-limit - lower-limit) * 0.05

These three cases are numbered cases 1, 2, and 3 respectively. This heuristic rule is shown in the following:

```
Rule-Initial-Cases
 (If    no cases exist

 Then (assert
        (case 1 mid-range-1 mid-range-2 ------ mid-range-n))
       (assert
        (case bottom-1 bottom-2 --------- bottom-n))
       (assert
        (case 3 top-1 top-2 ----------- top-n))
       (assert
        (basket 1 2 3))
       (assert
        (nextcase 4)))
```

If more cases are needed, this rule can be modified to incorporate any desired combinations. In this rule, a basket is asserted to notify that a set of cases has been generated and is ready for coverage analysis. The statement of "nextcase" indicates that the case number of next new case will be 4.

7.2 COVERAGE ANALYSIS AND GOODNESS EVALUATION

As indicated above, the objective of the test case generation is to cover as many branches as possible. When a new set of cases is generated, it is analyzed to determine what branches these cases have covered. This process is called coverage analysis and is performed by an instrumented simulator of the target software package. After the coverage analysis, it will be possible to determine what branches have not yet been covered and still need further cases.

The instrumented simulator of a target package functions just like the target package except it contains extra code to record the branch coverage of each condition and to calculate how "close" a test case has been to the condition boundary. For example, an IF statement:

```
IF (3*x+y) >= 21 THEN do-1 ELSE do-2
```

will be instrumented as

```
IF analyze((3*x+y) >= 21) THEN do-1 ELSE do-2
```

Here, "analyze" is a function defined in the simulator to perform coverage analysis and other evaluation tasks. The coverage analysis is basically a table filling process which records the covered branches. Assume that a test case (x=5, y=2) is analyzed. Since the value of the evaluated left-hand-side (LHS), i.e., 17, is smaller than the right-hand-side (RHS), i.e., 21, the FALSE or the ELSE branch will be executed. This means the ELSE branch would be covered.

In addition to providing coverage analysis, the simulator also calculates how close the LHS is from the RHS, based on the given test case. Here "closeness" is defined as

ABS(LHS - RHS) / MAX (ABS(LHS), ABS(RHS))

This measure tells the closeness of the test case to the condition boundary [DEA88]. When this measurement is small, it is generally true that this test case can be modified slightly to cover the other branch of the condition. In the previous example, with test case (x=5, y=2), LHS is 17 and RHS is 21. The closeness value is therefore (21 - 17) / 21 = 19%. Assume there is a second test case (x=6, y=2). Its closeness value is (21 - 20) / 21 = 4.8%. The second case will be considered "better" than the first case according to the criteria introduced above. This is because a smaller change on the second case may be enough to lead to the condition boundary or even beyond the boundary and cover the other branch.

The importance of the slight modification to the test case is based on the fact that the original case starts from the entry point and reaches the condition under consideration. Between these two points, this case is also used by other conditions. In order to reach the condition under consideration again and cover the other branch, the modified new case must pass through the same set of conditions and yield the same branching results. For this reason, the closeness of a test case (under a particular condition) can also be considered as its "goodness".

In the current version of system, the analyzer is written in C language. It is called as an external function from CLIPS.

In the TDG prototype, each condition contains two pieces of information. The first relates to the coverage of its branches. The second is the "best" test case that has reached this condition so far. If both branches of a condition have already been covered, this condition will not be considered for further test case generation. However, if only one branch is covered, more test cases will be generated, based on the best case. This heuristic is expressed in the following CLIPS rules, which have been modified slightly (from CLIPS) for easier understanding:

Rule-No-More-Case

```
If     (check ?cond)                    ;check condition ?cond
       (cond ?cond true true $?x)       ;both branches have been
                                        ;covered. Best case, $?x, is
                                        ;ignored.
then (assert (check (+ ?cond 1))))      ;check next condition.
```

Rule-More-Cases

```
(If    (check ?cond)                    ;check condition ?cond
       (not                             ;not both conditions are
       (cond ?cond true true ?x))       ;covered.  There  is  also  a
                                        ;best test case, ?x.
then (assert
       (generate-more ?cond ?x)))       ;pass the condition and the
                                        ;best case to the test case
                                        ;generator.
```

7.3 NEW TEST CASE GENERATION

The objective of this portion of case generation is to cover the branches that are not covered previously. As mentioned earlier, a request of this kind provides the condition to be considered and its best test case. The generation of more test cases should follow the general philosophy of modifying the best case by a small amount. With this guideline in mind, the test data generator (TDG) will produce several sets of new cases with various constraints.

Consider the best test case, case-i, of condition cond-i. Case-i is a list of numerical values: $(V_1, V_2, ..., V_j, ..., V_n)$. The j-th value in the list corresponds to the j-th input variable. Since it may happen that only part of the input variables would appear in a condition, the TDG needs to know which variables appear in the condition. The values of these variables have the direct impact of the branching decision and should be the candidates for modification. Let us assume $VL = (V_a, V_b, ...)$ contains these variables. The following heuristics have been used to generate new test cases:

1. Keep all variable values unchanged except one variable which is in VL. This would simplify the condition under consideration. For example, in the previous best case, (x=6, y=2), if we keep x unchanged, the condition can be simplified in the following steps:

```
a.        3 * x + y  >=  21      ;    x=6

b.           18 + y  >=  21

c.                y  >=  3
```

The last expression gives the condition boundary. Three more new cases can be generated, i.e., (x=6, y=3+e), (x=6, y=3), (x=6, y=3-e). Here, e is defined as a small positive number,

e.g., e = (upper-limit of y - lower-limit of y) / 100. The same method is then applied to every variable in VL. If there are m variables in VL, then 3 * m cases will be generated. The simplification process is performed in C- subroutines. This is because CLIPS does not provide symbolic evaluation functions.

2. Keep all variable values unchanged except two variables which are in VL. One of the variables is instantiated to its mid-range value. With this instantiation, only one variable is left in the condition. The simplification process previously mentioned can then be applied. Assume we are considering a condition:

   4 * x * y + 3 * x =< 9 * y

The best test case we have so far for this case is (x=2, y=15, z=10.6). Since only x and y appear in this condition, only their values need to be considered. Also assume the ranges for x and y are [-10.5, 20.5] and [-5.5, 30.5], respectively. Three new test cases can be generated in the following steps:

a. Assign y = mid-range of y = [30.5-(-5.5)]/2 = 18

b. Replace every appearance of y in the condition with 18.

   3 * x * 18 + 3 * x =< 9 * 18

   57 * x =< 162

   x =< 2.842

c. Generate 3 cases (x=2.842+e, y=18, z=10.5), (x=2.842, y=18, z=10.5), (x=2.842-e, y=18, z=10.5).

   The system then instantiates x to its mid-range and repeats the process.

These two heuristics will generate many new cases. Additional branch coverage normally can be achieved. However, it may happen that new cases never reach the target condition due to the modification of previous cases. If this happens, the best test case of the target condition would not have been changed. Since the generation of new test cases is based on the best case, the same set of test cases will be generated over and over again and they will never out perform the original best case. The TDG needs a mechanism to prevent this. We define a "bag" to be associated with each condition. This bag records all best cases that the system has used to generated cases for the condition. As more cases are generated and analyzed, the best case of each condition will evolve. This is the reason that there may be more than one best case in a bag. When the best test case of a condition is found to have been used before, an unnecessary loop may exist. This indicates that previous heuristics did not yield good cases. If this happens, the following third and the fourth heuristics will be applied:

3. If there is only one variable in VL, then modify all variable values except the one in VL by 10% of their ranges and then apply the simplification process described in the first

heuristic to find the condition. In a condition with only one variable, the only way of covering both branches is to modify this variable's value according to simplification process of the first heuristic. This is required no matter how a test case reaches this condition. If the modification causes the change of the coverage path (i.e., the condition under consideration can no longer be reached), other variable values must be modified to compensate for the change. However, the task of figuring out how a modified variable value would impact the branchings of other conditions and how other variables' values should be modified to compensate for this impact is still a future study topic. Currently, all other variables' values are changed by 10 percent of their correspondent ranges to compensate for this impact. After the modification to all variables except the one in which the condition is made, the simplification step in heuristic 1 is applied to determine its values.

4. If there is more than one variable in VL, heuristic 2 will be applied, except all other variables will be modified by 10 percent of their ranges.

7.4 TEST DATA GENERATION EXAMPLES

In this section, a simulation of the test data generation process is presented to illustrate the functions of the system. Assume the flow chart of a target software package is given in Figure 5. There are three input variables to the package, i.e., x, y, and z. They are all real numbers and have ranges, [30, 200], [-220, 20], and [-100, 312] respectively. The expressions of the conditions are:

Cond-1: z + 20 < 3 * x

Cond-2: 3 * x * y =< 4 * y

Cond-3: z > y + 100

When this information is presented to the TDG, three initial test cases will be generated based on the first heuristic. They are the mid-range case, (case 1, 115, -100, 106), the bottom-range case, (case 2, 38.5, -208, -79.4), and the top-range case, (case 3, 191.5, 8, 291.4). These cases are then fed to the coverage analyzer. The conditions and branches that each case reaches are:

Case-1: (cond-1 True), (cond-2 True)

Case-2: (cond-1 True), (cond-2 True)

Case-3: (cond-1 True), (cond-2 False)

It can be seen that all cases went through the truth branch of cond-1, two cases went through the truth branch of cond-2, and one case went through the false branch of cond-2. The coverage table at this point is shown in Figure 6. Based on the goodness of a case associated with a condition defined earlier, the best test cases for cond-1 is case-3 and the best test case for cond-2 is case-2. This information is sent back to the TDG. The TDG will start by checking cond-1. Since only one branch of cond-1 is covered, more cases should be generated for cond-1. The first and the second heuristics are applied and case-3 is used as the best case. This will generate six new cases. The case generation sequence is:

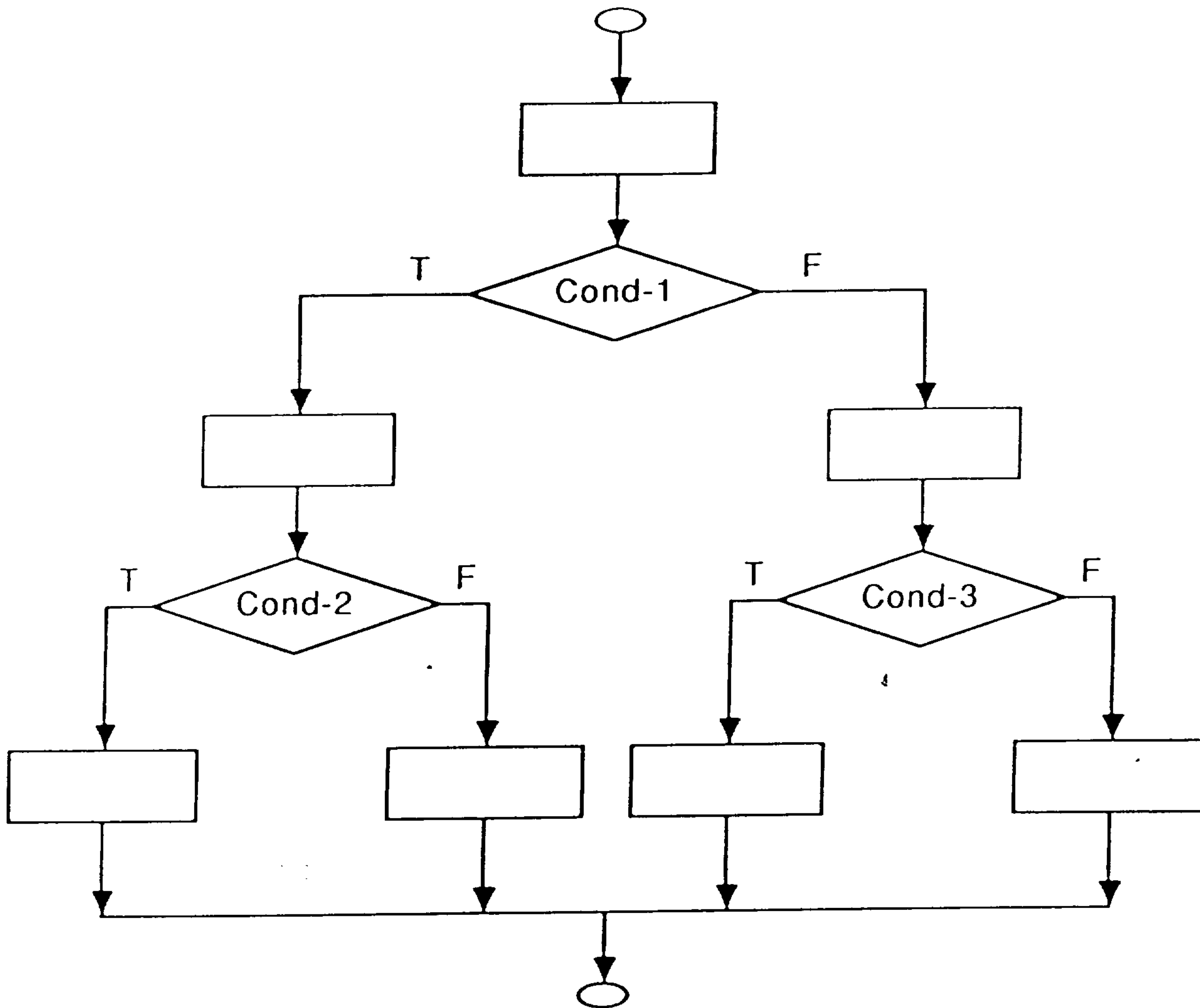


Figure 5 - Example Target Software Package Flowchart

Condition	Branch T	Branch F
1	X	
2	X	X
3		

Figure 6 - Partial Coverage Table

(case 4,191.5, 8, 554.5) ;keep x unchanged, modify z only,
;z is out of range, illegal case.

(case 4, 103.8, 8, 291.4) ;keep z unchanged, modify x only

(case 5, 102.1, 8, 291.4) ;keep z unchanged, modify x only

(case 6, 105.5, 8, 291.4) ;keep z unchanged, modify x only

(case 7, 115, 8, 425) ;keep x at mid-range, modify z only
;z is out of range, illegal case

(case 7, 42, 8, 106) ;keep z at mid-range, modify x only

(case 8, 40.3, 8, 106) ;keep z at mid-range, modify x only

(case 9, 43.7, 8, 106) ;keep z at mid-range, modify x only

The process then checks cond-2. Since both branches of cond-2 are covered, no more cases are needed. When cond-3 is checked, it is found that it has never been reached (or covered) before. No cases will be generated for cond-3 at this point. The process then passes the newly generated six cases to the coverage analyzer. The analysis result will be:

case-4: (cond-1 False), (cond-3 True)

case-5: (cond-1 False), (cond-3 True)

case-6: (cond-1 True), (cond-2 False)

case-7: (cond-1 False), (cond-3 False)

case-8: (cond-1 False), (cond-3 False)

case-9: (cond-1 True), (cond-2 False)

At this point, all conditions are fully covered. The test data generator will stop. In all, 9 cases were generated.

7.5 SYSTEM INTERFACE MECHANISM

The technical description given above tends to obscure the interactions of the Test Data Generator (TDG) with the rest of the system. This section is intended to clarify the mechanisms by which this is accomplished.

The TDG will only respond to feedback information from the Test Execution Module (TEM) and the Test Coverage Analysis (TCA) component. However, it should be clear that these two modules cannot function without some original test cases being supplied. It is expected that user-supplied test cases will be part of any good Ada software

design. The QUEST design accommodates these by allowing them to be input first, i.e., prior to automatically generating additional test cases.

After a packet of test data is generated, a round of executions of this data will follow. Updated TEM and TCA information will then be returned to TDG in order to prepare for the next round of test data generation. After each round the test cases added to the file will be marked according to the round in which they were generated.

For purposes of efficient verification and regression testing it might be beneficial to indicate a priority on the tests. It is expected that TDG will generate hundreds or even thousands of tests for a given module. Depending upon the automated comparison capability, it may not be possible to verify every one of these against an independent execution of the design. This being the case, the following priority scheme is suggested:

0 - user defined test cases (highest);

1 - first test cases to add to control coverage; these along with the 0-priority cases will form a minimal test set;

2 - subsequent n test cases which do not add to control coverage but provide additional data coverage, where n is a value dependent upon the program characteristics;

3 - this is the lowest priority, and it would be assigned to any test case not falling in the three given above.

8.0 TEST COVERAGE ANALYZER

When the user selects option 1 (Begin Testing) from the Automatic Testing Menu, the Test Coverage Analyzer (TCA) is invoked. The TCA receives data about the module to test and the maximum number of test packets that the TDG should generate.

For each test packet, the TCA invokes the Test Data Generator. The instrumented Ada module is then executed using the newly generated data. The intermediate data created during the execution is first accumulated for the printed reports, and then it is examined by the TCA.

The TCA creates a coverage list containing an entry for each decision encountered so far, and it keeps up with whether the decision has been covered for TRUE and for FALSE. After all of the new intermediate results have been placed into the coverage list, the TCA makes assertions to the TDG.

These assertions allow the TDG to select input to cover all currently uncovered decisions in the Ada module. The TCA informs the TDG of each decision in the Ada module that has not been completely covered in testing, and it keeps up with the percentage of decisions that have been completely covered.

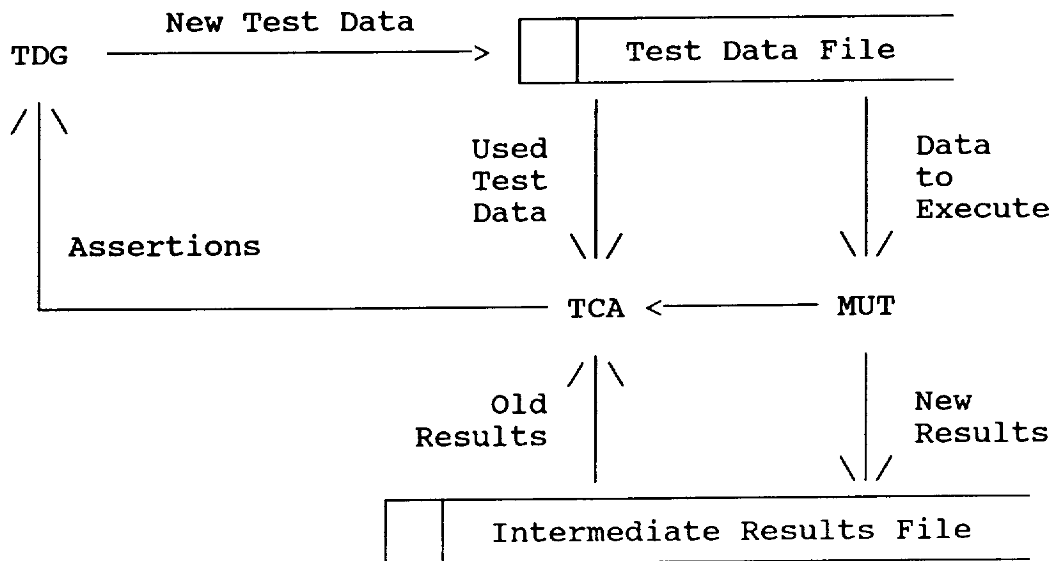


Figure 7. Test Coverage Analyzer

9.0 RECOMMENDATIONS FOR FUTURE RESEARCH

The completion of Phase I of this project resulted in a working prototype which proved the concept of expert-system assisted test case generation to increase software reliability. 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 expert system tool, CLIPS, were used as a basis for the expert system. Given this prototype it is now possible to evaluate various rule base approaches such that an improved set of rules can be developed and validated.

Appendix F presents the proposal for Phase II of this project. It will extend the testing tool developments presented above as well as incorporating the development of reverse-engineering tools. In the area of test-tool innovation, the following goals have been established for Phase II: (1) to continue to develop and improve the current user interface to support the other goals of the research effort, (2) to 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. These compose Task 1 of Phase 2, and they are discussed in detail in Appendix F.

Task 2 of Phase 2 involves the development of reverse engineering tools, basically in the form of graphical representations. The goals of this task are: (1) to study, formulate and evaluate graphical representations for Ada software, (2) to develop a prototype reverse engineering tool that includes support for generation of both algorithmic and hierarchical diagrams, and (3) to investigate the generation of additional graphical representations to provide task, package, and data flow views of Ada software. The details and scheduled activities for this task are also given in Appendix F.

10. REFERENCES

[ADR82] Adrion, W. Richards, et al., "Validation, Verification, and Testing of Computer Software", **ACM Computing Surveys** Vol. 14, June 1982.

*[AHO85] Aho, A. V., Sethi, R. and Ullman, J.D., **Compilers, Principles, Techniques, and Tools**, Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

[BEI83] Beizer, B., **Software Testing Techniques**, New York: Van Nostrand Reinhold Company, 1983.

[BEI84] Beizer, B., **Software System Testing and Quality Assurance**, New York: Van Nostrand Reinhold Company, 1984.

[BOE75] Boehm, B. W., et al., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software", IEEE Trans. on Software Engineering, Vol. SE-1, March, 1975.

[BOU85] Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M. C., "Application of Prolog to Test Sets Generation from Algebraic Specifications", **TAPSOFT Joint Conference on Theory and Practice of Software Development**, March 1985.

[BRO86a] Brown, D. B., Haga, Kevin D., and Weyrich, Orville, Jr., "QUEST - Query Utility Environment for Software Testing", **International Test and Evaluation Association 1986 Symposium Proceedings**, pp. 38-43.

[BRO86b] Brown, D. B., "Test Case Generator for TIR Programs", Contract Number DAAH01-84-D-A030 Final Report, September 30, 1986.

[BRO87] Brown, D. B., "Advanced Simulation Support", Contract Number DAAHOI-84-A030/0006 Final Report, June 17, 1987.

[CER81] Ceriani, M., Cicu, A., and Maiocchi, M., "A Methodology for Accurate Software Test Specification and Auditing", in **Computer Program Testing**, 1981.

[CHO86] Choquet, N., "Test Data Generation Using a Prolog with Constraints", in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, July 1986.

[CLA76] Clarke, Lori A., "A System to Generate Test Data and Symbolically Execute Programs", **IEEE Transactions on Software Engineering**, Vol. SE-2, pp. 215-222, September 1976.

---

*Reference not discussed in Section 2.

[CLA86] Clarke, L. A., Podgurski, A., Richardson, D. J. and Zeil, S. J., "An Investigation of Data Flow Path Selection Criteria", **Proc. Workshop on Software Testing,** IEEE Computer Society Press, July 1986.

[CLI87] CLIPS Reference Manual, Version 4.1, Artificial Intelligence Section, Johnson Space Center, NASA, September 1987.

[DEA88] Deason, William H., Rule-Based Software Test Data Generation, MS Thesis, Department of Computer Science and Engineering, Auburn University, December, 1988.

[DIL88] Dillion, L. K., "Symbolic Execution-Based Verification of Ada Tasking Programs", **3rd International IEEE Conference on Ada Applications and Environments**, May, 1988.

[DEM78] DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Hints on Test Data Selection: Help for the Practicing Programmer", **IEEE Computer**, Vol. 11, No. 4, April 1978.

[DEU82] Dentsch, M. S., Software Verification and Validation, Englewood Cliffs, NJ, Prentice-Hall Inc., 1982.

[DUR80] Duran, J. W. and Wiorkowski, J. J., "Quantifying Software Validity by Sampling", **IEEE Transactions on Reliability**, Vol. R-29, No. 2, June 1980.

[DUR81] Duran, J. W. and Ntafos, S., "A Report on Random Testing", in **Proceedings of the 5th International Conference on Software Engineering**, March 9-12, 1981.

[DUR84] Duran, J. W. and Ntafos, S., "An Evaluation of Random Testing", **IEEE Transactions on Software Engineering,** Vol. SE-10, pp. 438-444, July 1984.

*[FAI85] Fairley, R. E., **Software Engineering Concepts**, McGraw-Hill, New York, 1985.

[FIS88] Fisher, A. S., **CASE - Using Software Development Tools**, John Wiley & Sons, Inc., New York, 1988.

[FOS80] Foster, K. A., "Error Sensitive Test Case Analysis (ESTCA)", **IEEE Transactions on Software Engineering**, Vol. SE-6, pp. 258-264, May 1980.

[FRA86] Frankl, P. G., and Weyuker, E. J., "Data Flow Testing in the Presence of Unexecutable Paths", in **Proc. Workshop on Software Testing,** IEEE Computer Society Press, July 1986.

[FRA88] Frankl, P. G., and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", IEEE Trans on Software Engineering, Vol. 14, No. 10, October 1988.

[GIR86] Girgis, M. R., and Woodward, M. R., "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria", in **Proc. Workshop on Software Testing,** IEEE Computer Society Press, July 1986.

[GLA81] Glass, Robert L., "Persistent Software Errors", **IEEE Transactions on Software Engineering**, Vol. SE-7, pp. 162-168, March 1981.

[GOO75] Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection", **IEEE Transactions on Software Engineering**, Vol. SE-1, No. 2, June 1975.

[GOR86] Gordon, A. J., and Finkel, R. A., "TAP: A Tool to Find Timing Errors in Distributed Programs", in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, July 1986.

[HOW75] Howden, W. E., "Methodology for the Generation of Program Test Data",**IEEE Transactions on Software Engineering**, Vol. C-24, May 1975.

[HOW76] Howden, W. E.,"Reliability of the Path Analysis Testing Strategy", **IEEE Transactions on Software Engineering**, Vol. SE-2, September 1976.

[HOW78a] Howden, W. E., "Theoretical and Empirical Studies of Program Testing", **IEEE Transactions on Software Engineering**, Vol. SE-4, July 1978.

[HOW78b] Howden, W. E., "DISSECT - A Symbolic Evaluation and Program Testing System", **IEEE Transactions on Software Engineering**, Vol. SE-4, January 1978.

[HOW80] Howden, W. E., "Functional Program Testing", **IEEE Transactions on Software Engineering**, Vol. SE-6, March 1980.

[HOW81] Howden, W. E., "Errors, Design Properties, and Functional Program Testing", in **Computer Program Testing**, 1981.

[HOW82a] Howden, W. E., "Life-Cycle Software Validation", **IEEE Computer**, Vol. 15, No. 2, February 1982.

[HOW82b] Howden, W. E., "Weak Mutation Testing and Completeness of Test Sets", **IEEE Transactions on Software Engineering**, Vol. SE-8, July 1982.

[HOW86] Howden, W. E., "A Functional Approach to Program Testing and Analysis", **IEEE Transactions on Software Engineering**, Vol. SE-12, October 1986.

[HOW87] Howden, W. E., **Functional Program Testing and Analysis**, McGraw-Hill, New York, 1987.

[HUA75] Huang, J. C., "An Approach to Program Testing", **ACM Computing Surveys**, Vol. 7, September 1975.

[HUA78] Huang, J. C., "Program Instrumentation and Software Testing", **IEEE Computer**, Vol. 11, No. 4, April 1978.

[LAS83] Laski, J. W., and Korel, B., "A Data Flow Oriented Program Testing Strategy", **IEEE Transactions on Software Engineering**, Vol. SE-9, May 1983.

[MEY79] Myers, G. J., **The Art of Software Testing**, New York: John-Wiley & Sons, 1979.

[MIL84] Miller, E. F., "Software Testing Technology: An Overview", in **Handbook of Software Engineering**, New York: Van Nostrand Reinhold Company, 1984.

[NTA79] Ntafos, S. C. and Hakimi, S. L., "On Path Coverage Problems in Digraphs and Applications to Program Testing", **IEEE Transactions on Software Engineering**, Vol. SE-5, September 1979.

[NTA84] Ntafos, S. C., "On Required Element Testing", **IEEE Transactions on Software Engineering**, Vol. SE-10, November 1984.

*[NTA88] Ntafos, S. C.,"A Comparison of Some Structural Testing Strategies", **IEEE Transactions on Software Engineering**, Vol. 14, June 1988.

[OST79] Ostrand, T. J. and Weyuker, E. J., "Error-Based Testing", in **Proc. 1979 Conf. Inf. Sciences and Systems**, 1979.

[OST86] Ostrand, T. J., Sigal, R., and Weyuker, E. J.,"Design for a Tool to Manage Specification-Based Testing", in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, 1986.

[PAN78] Panzl, D. J.,"Automatic Software Test Drivers", **IEEE Computer**, Vol. 11, No. 4, April 1978.

[PRA87] Prather, R. E. and Myers, J. P., Jr., "The Path Prefix Software Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987.

[RAM66] Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Considerations", **Journal of the ACM**, Vol. 13, April 1966.

[RAM75] Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems", **IEEE Transactions on Software Engineering**, Vol. SE-1, March 1975.

[RAM76] Ramamoorthy, C. V. et al., "On the Automated Generation of Program Test Data", **IEEE Transactions on Software Engineering**, Vol. SE-2, December 1976.

[RAP85] Rapps, S. and Weyuker, E. J., "Selecting Software Test Data Using Data Flow Information", **IEEE Transactions on Software Engineering**, Vol. SE-11, No. 4, April 1985.

[RED83] Redwine, S. T., Jr., "An Engineering Approach to Software Test Data Design", **IEEE Transactions on Software Engineering**, Vol. SE-9, March 1983.

[ROS85A] Ross, S. M., "Statistical Estimation of Software Reliability", **IEEE Transactions on Software Engineering**, Vol. SE-1, No. 5, May 1985.

[ROS85B] Ross, S. M., "Software Reliability: The Stopping Rule Problem", **IEEE Transactions on Software Engineering**, Vol. SE-11, No. 12, December 1985.

[RUB75] Rubey, R. J., et al., "Quantitative Aspects of Software Validation", **IEEE Transactions on Software Engineering**, Vol. SE-1, June 1975.

[SHO83] Shooman, M. L., **Software Engineering**, New York: McGraw- Hill Book Company, 1983.

[SNE86] Sneed, H. M., "Data Coverage Measurement in Program Testing", in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, July 1986.

[SOL85] Solis, D. M., "AutoParts - A Tool to Aid in Equivalence Partition Testing", in **Proc. SoftfairII: Second Conf. Software Development Tools, Techniques, and Alternatives**, 1985.

[TAI80] Tai, K. C., "Program Testing Complexity and Test Criteria," IEEE Trans on Software Engineering, Vol. SE-6, pp 531-538, November 1980.

[TAY86] Taylor, R. N., and Kelly, C. D., "Structural Testing of Concurrent Programs", in **Proc. Workshop on Software Testing,** IEEE Computer Society Press, July 1986.

[TSA86] Tsalalikhin, L., "Function of One Unit Test Facility", in **Proc. Workshop on Software Testing,** IEEE Computer Society Press, July 1986.

[VIC84] Vick, C. R., and Ramamoorthy, C. V., **Handbook of Software Engineering**, New York: Van Nostrand Reinhold Company Inc., 1984.

[VOG80] Voges, Vdo, et al, "SADAT-An Automated Testing Tool," IEEE Trans. on Software Engineering, Vol. SE-6, May 1980.

[VOG85] Voges, U. and Taylor, J. R., "Systematic Testing", in **Verification and Validation of Real-Time Software**, Ed. by W. J. Quirk, New York: Springer-Verlag, 1985.

[VOU88] Vouk, Mladen A., McAllister, David F., and Tai, K. C., "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software", in **Workshop on Software Testing Proceedings**, IEEE Computer Press, 1986.

[WEY80] Weyuker, E. J. and Ostrand, T. J., "Theories of Testing and the Application of Revealing Subdomains", **IEEE Transactions on Software Engineering,** Vol. SE-6, May 1980.

[WEY86] Weyuker, E. J., "Axiomatizing Software Test Data Adequacy", **IEEE Transactions on Software Engineering,** Vol. SE-12, No. 12, December 1986.

[WEY88a] Weyrich, O. R., Jr., Brown, D. B., and Miller, J. A., "The Use of Simulation and Prototypes in Software Testing", in **Tools for the Simulation Profession -** Proceedings of the 1988 Conferences, Orlando, Florida, Society for Computer Simulation.

[WEY88b] Weyrich, O. R., Jr., Cepeda, S. L., and Brown, D. B., "Glass Box Testing Without Explicit Path Predicate Formation", 26th Ann. Conf. Southeast Regional ACM, April 20-22, 1988, Mobile, Alabama.

[WHI80] White, Lee J. and Cohen, E. I., "A Domain Strategy for Computer Program Testing", **IEEE Transactions on Software Engineering**, Vol. SE-6, May 1980.

[WHI86] White, L. J., and Perera, I. A., "An Alternative Measure for Error Analysis of the Domain Testing Strategy", in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, July 1986.

[WIL85] Wilson, C. and Osterweil, L. J., "Omega - A Data Flow Analysis Tool for the C Programming Language", **IEEE Transactions on Software Engineering**, Vol. SE-11, No. 9, September 1985.

[W0080] Woodward, M. R., et al., "Experience with Path Analysis and Testing of Programs", **IEEE Transactions on Software Engineering**, Vol. SE-6, May 1980.

[YOU86] Young, M., and Taylor, R. N., "Combining Static Concurrency Analysis with Symbolic Execution" in **Proc. Workshop on Software Testing**, IEEE Computer Society Press, July 1986.

APPENDIX A

QUEST/ADA IORL SYSTEM SPECIFICATION

This appendix contains the IORL specifications for the QUEST/Ada system. A brief explanation related to the interpretation of IORL is in order (for details, obtain the IORL Reference Manual, Teledyne Brown Engineering, Inc., 1984). IORL specifications are arranged into sections. The section types used for the QUEST/Ada system include:

SBD - Schematic Block Diagram,

IORTD - Input Output Relationships and Timing Diagram, and

PPD - Predefined Process Diagram.

The SBDs are purely structural diagrams showing the capacity for data flow. The links on these diagrams are called interfaces, which show how data may flow between the various blocks, which are properly called components. Components have the capacity to operate concurrently.

Each component has a procedure by which it turns its input interface data into data to be transmitted over the output interface. The IORTD is the highest level of control flow for a component. IORTD-x is the sole high-level procedural diagram for component x in the SBD. It usually abstracts the many detailed innerworkings of a component into a few input, process, and output symbols. These symbols, on the IORTD, are connected by control flow indicators which show transfer of control, not dataflow (as in the SBD).

The double-edged rectangle within the IORTD (or PPD) section indicates the abstraction of more detailed control flow contained in the appropriately numbered PPD section. Since PPDs may themselves contain reference to other PPDs, IORL supports stepwise refinement and top-down design. More importantly, every effort has been made to organize and group sequences of events within PPDs such that a complete thought unit is on one page. Therefore, the IORL specification should be read sequentially without a great deal of referral between pages. Each page contains one thought unit which should be mastered before proceeding to the next page.

The first two diagrams are the SBDs which were included and discussed in Section 3. They are repeated here for completeness. Note that the "DOC" field of the identification fields (bottom of diagram) shows the first of these to be QUEST-ADA, the same as the system name for the highest level SBD. The second has DOC:QA, which indicates that component QA on the previous SBD is being analyzed into its respective components. In this SBD the

being analyzed into its respective components. In this SBD the dotted interfaces are external, in this case linking to the user.

Each component in the SBD for DOC:QA is analyzed by an IORTD. The IORTD numbers correspond to the component number. Thus, DOC:QA; IORTD-1 is a control flow analysis of the Parser/Scanner. We have chosen to place the PPD sections behind the respective calling IORTD/PPD sections. Thus, since IORTD-1 references PPDs 10100 and 10200, they follow immediately. PPD-10200 references PPD-10220 so it is next. PPDs referenced but not elaborated are either still in design or else they are considered to be of low enough specification to be programmed. Ultimately all of the lowest level PPDs will have direct references to their respective source code files.

Note that IORTD-2 of DOC:QA (the Test Data Generator) follows the sections for IORTD-1. Its PPDs are numbered in the 20000 series, and the single one elaborated follows. Similarly, the Test Execution Module (IORTD-3) and the Test Coverage Analysis (IORTD-4) follow. As additional details of the design evolve, they will be added in their corresponding positions to maintain a logical presentation of the system.

Source Code
TDG Control Parameters
Initial/Updated User Test Data
Regression Test Signal

QUEST_ADA-12

1
USER
Person responsible for unit-level
testing of Ada code

2
QA
Query Utility Environment
for Software Testing - Ada
(QUEST/Ada)

QUEST_ADA-21

Coverage Analysis Reports
Source Code Listing
Test Case Execution Results

SYSTEM: QUEST_ADA CLASS: DOC: QUEST_ADA SEC: SBD PAGE: 1
MAP: DEFAULT_MAP ISSUE: D DATE: 17-Jul-89

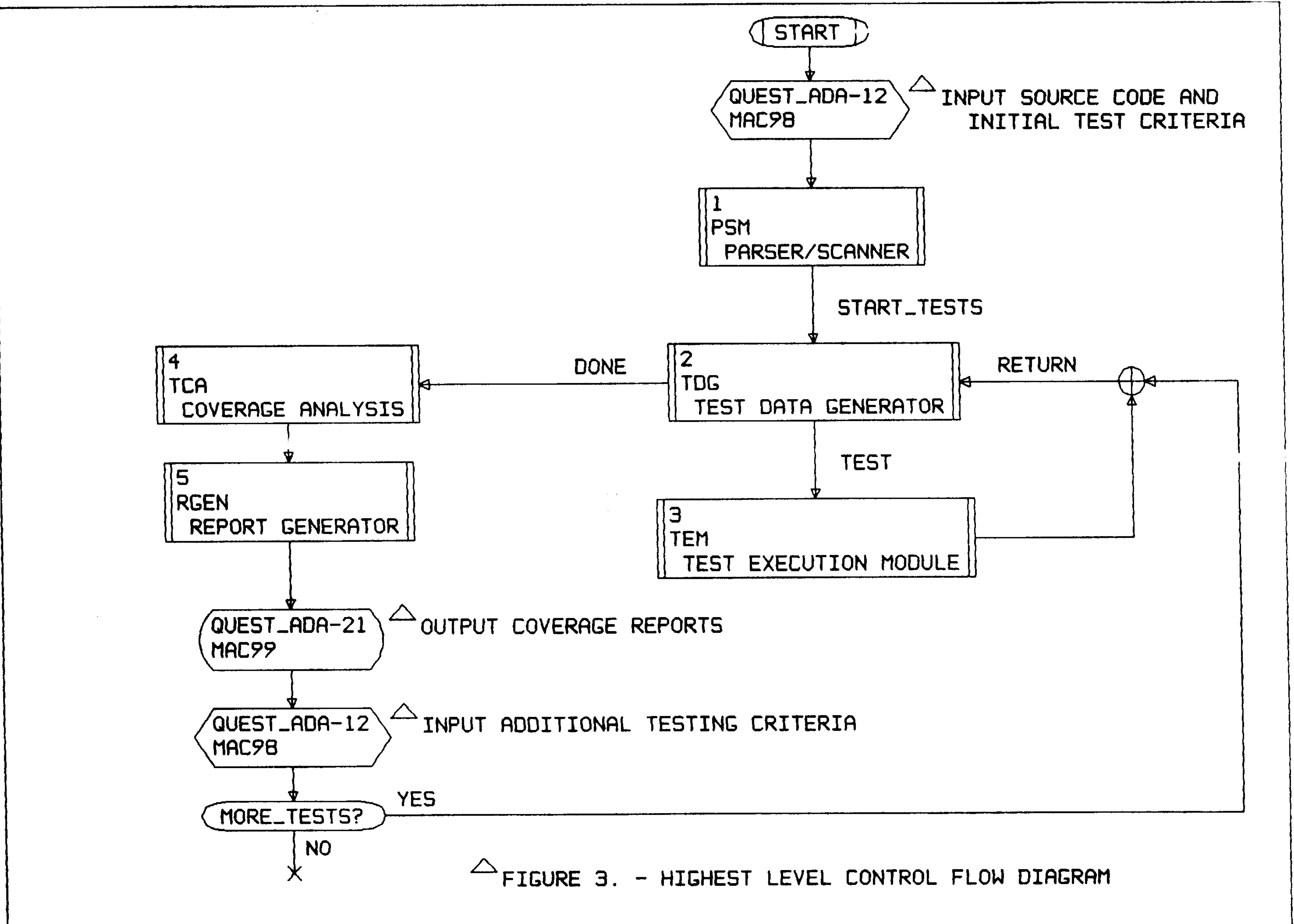
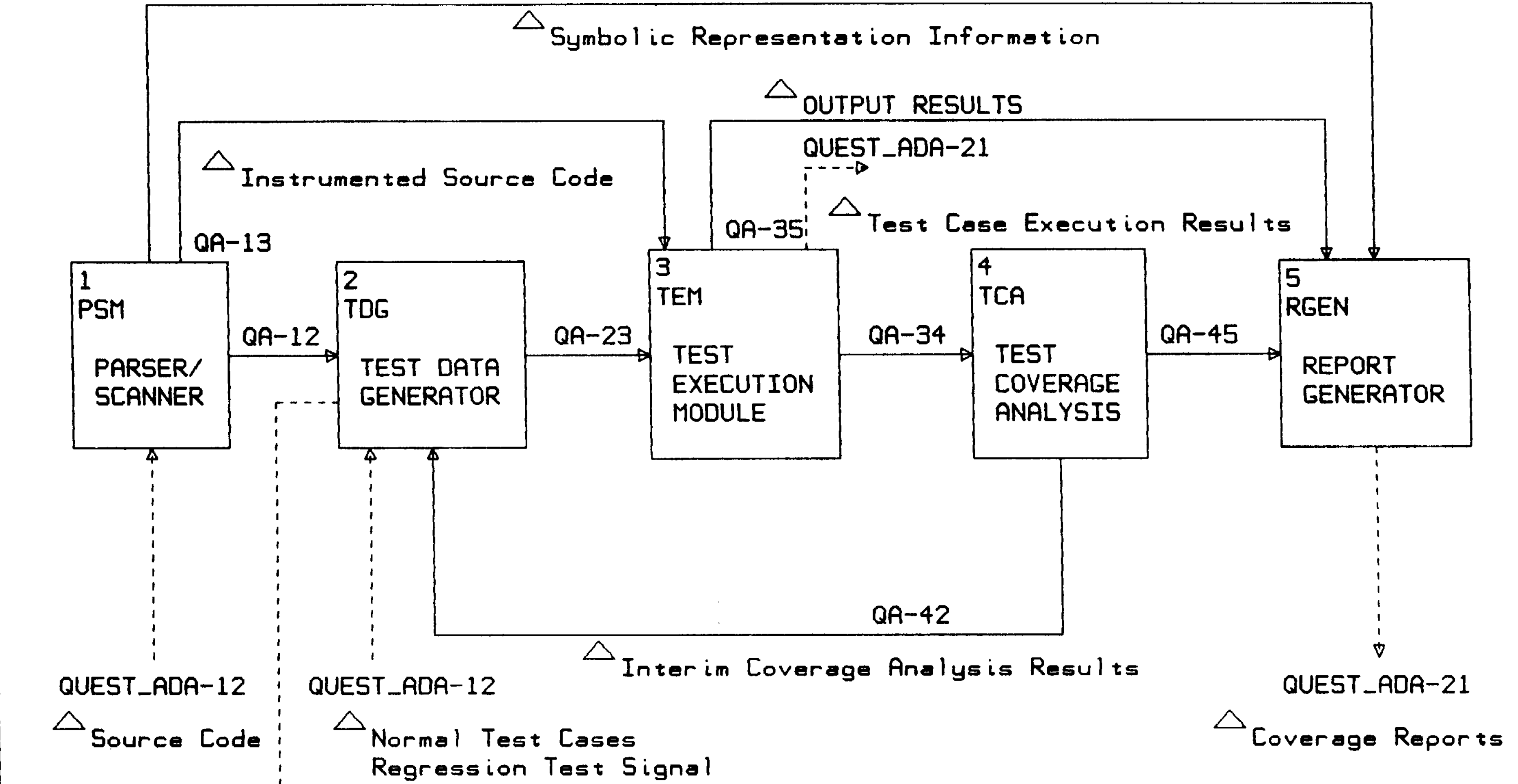
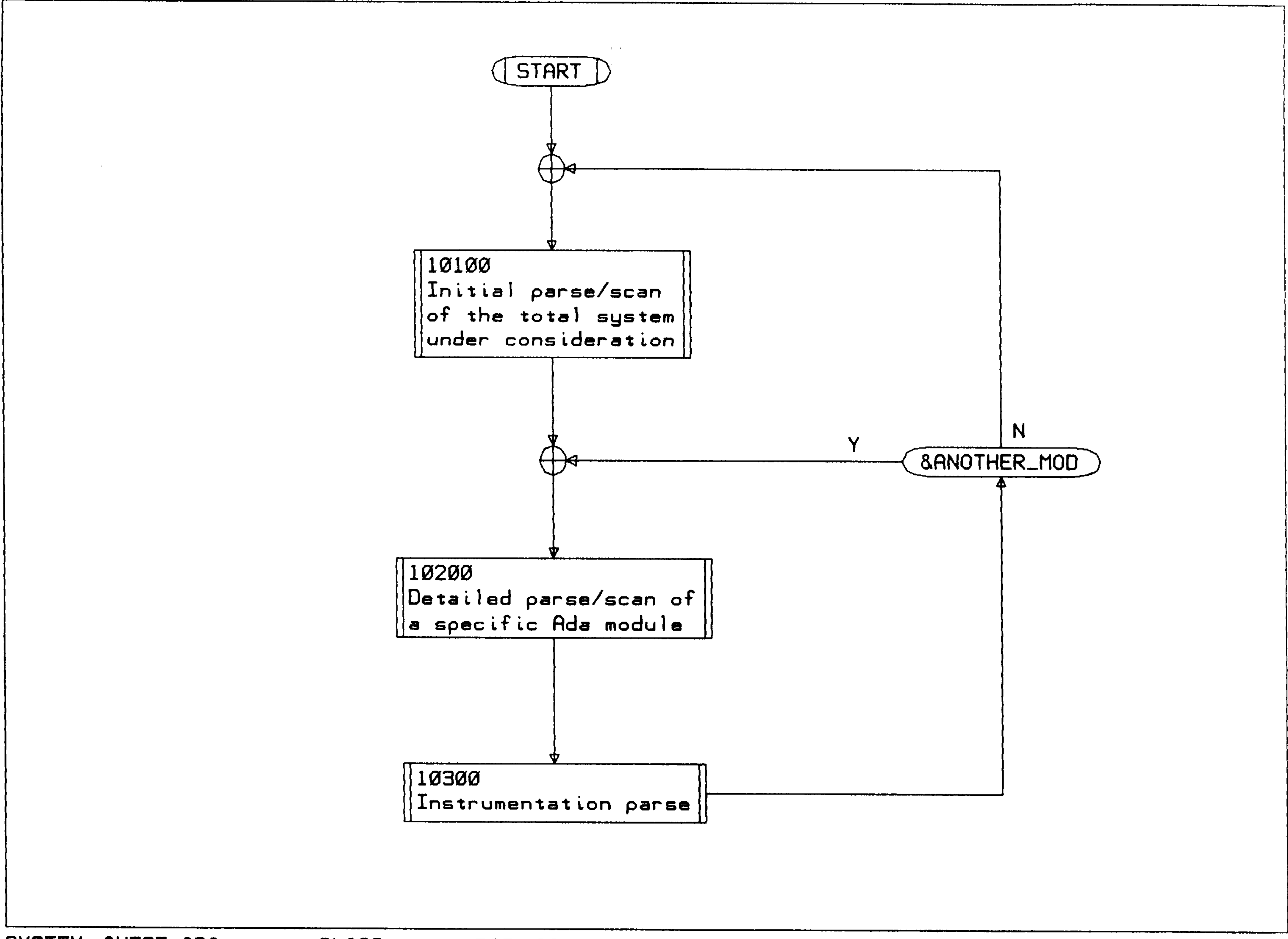
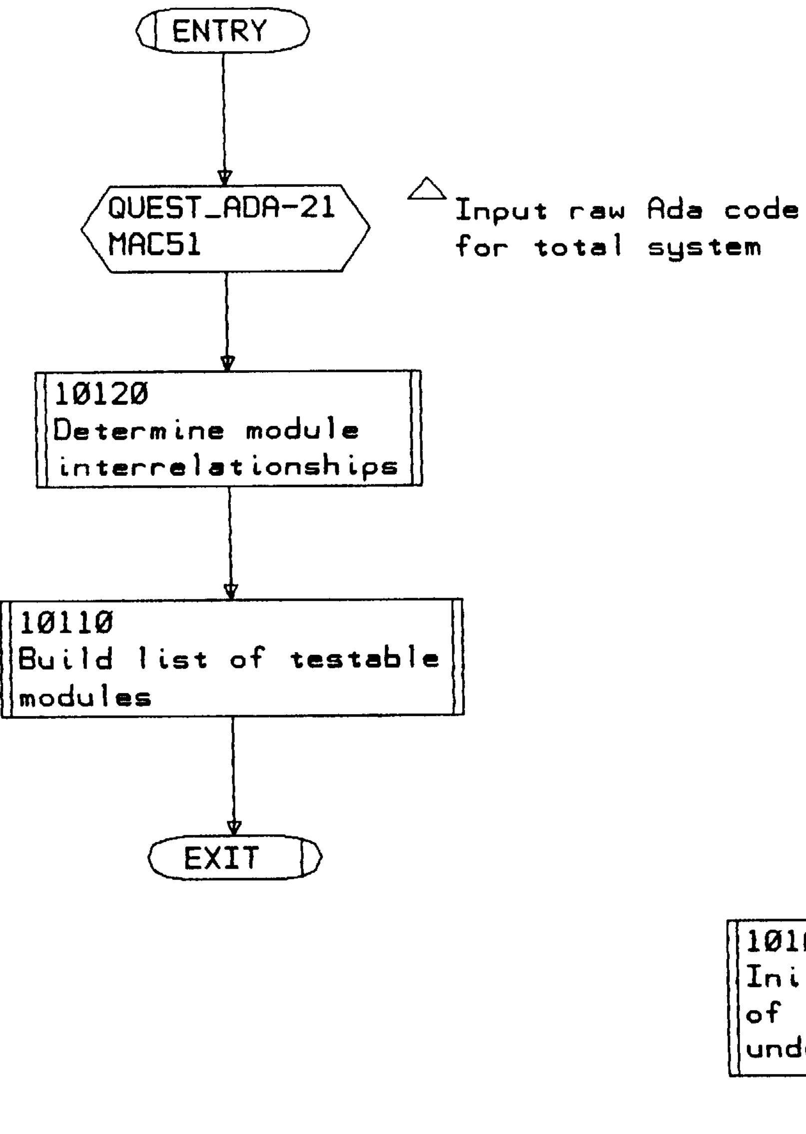


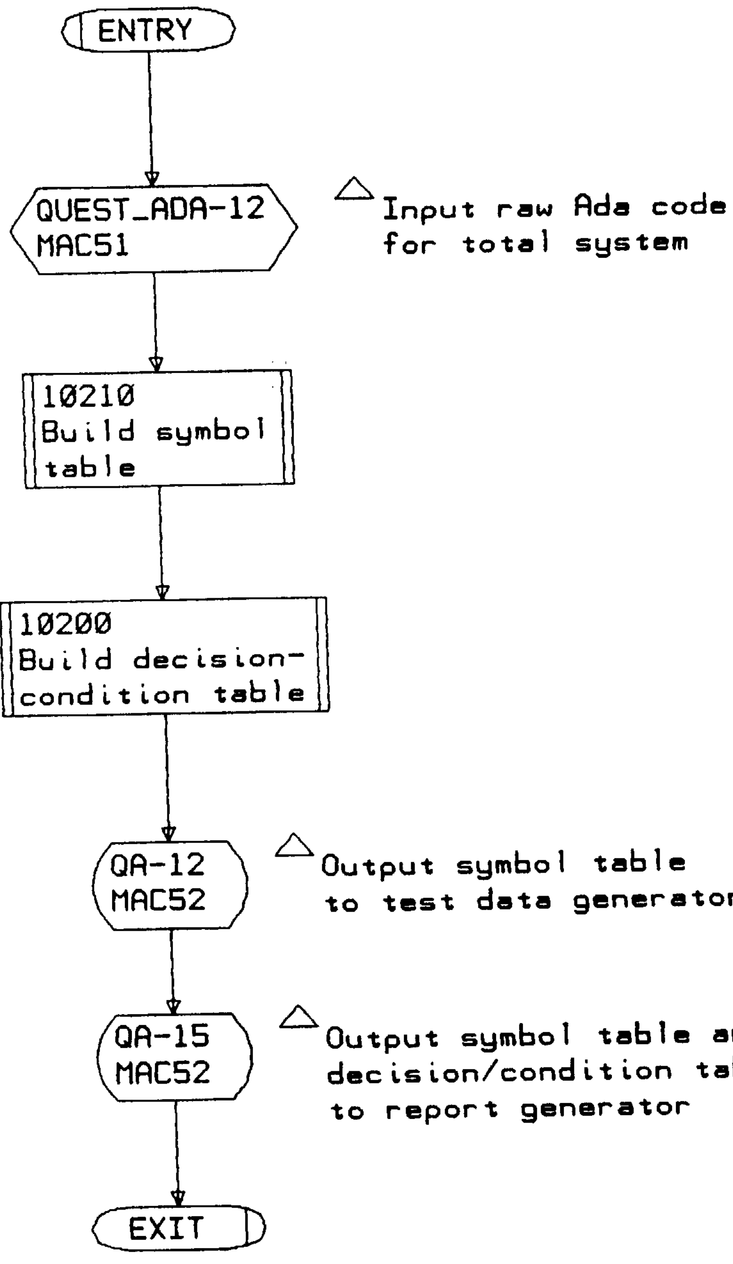
FIGURE 3. - HIGHEST LEVEL CONTROL FLOW DIAGRAM


QA-12: Symbolic Representation Information
QA-23: Test Case Number, Test Data
QA-34: Test Execution Results
QA-15 C QA-13
QA-15
Symbolic Representation Information
OUTPUT RESULTS
QUEST_ADA-21
Instrumented Source Code
QA-35
Test Case Execution Results
QA-13
1
PSM
PARSER/
SCANNER
QA-12
2
TDG
TEST DATA
GENERATOR
QA-23
3
TEM
TEST
EXECUTION
MODULE
QA-34
4
TCA
TEST
COVERAGE
ANALYSIS
QA-45
5
RGEN
REPORT
GENERATOR
QA-42
Interim Coverage Analysis Results
QUEST_ADA-12
Source Code
QUEST_ADA-12
Normal Test Cases
Regression Test Signal
QUEST_ADA-21
Coverage Reports
QUEST_ADA-21
Dynamic
coverage information
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: SBD
PAGE: 1
DATE: 17-Jul-89


START
10100
Initial parse/scan
of the total system
under consideration
10200
Detailed parse/scan of
a specific Ada module
10300
Instrumentation parse
&ANOTHER_MOD
Y
N
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: IORTD-1
PAGE: 1
DATE: 5-Jul-89



SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: PPD-10100
PAGE: 1
DATE: 12-Jun-89



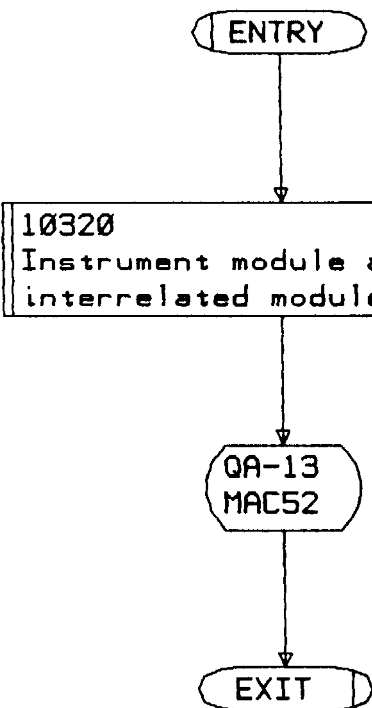
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP

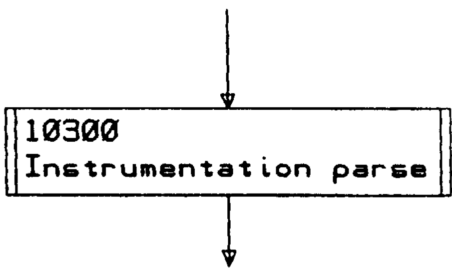
CLASS:

DOC: QA
ISSUE: D

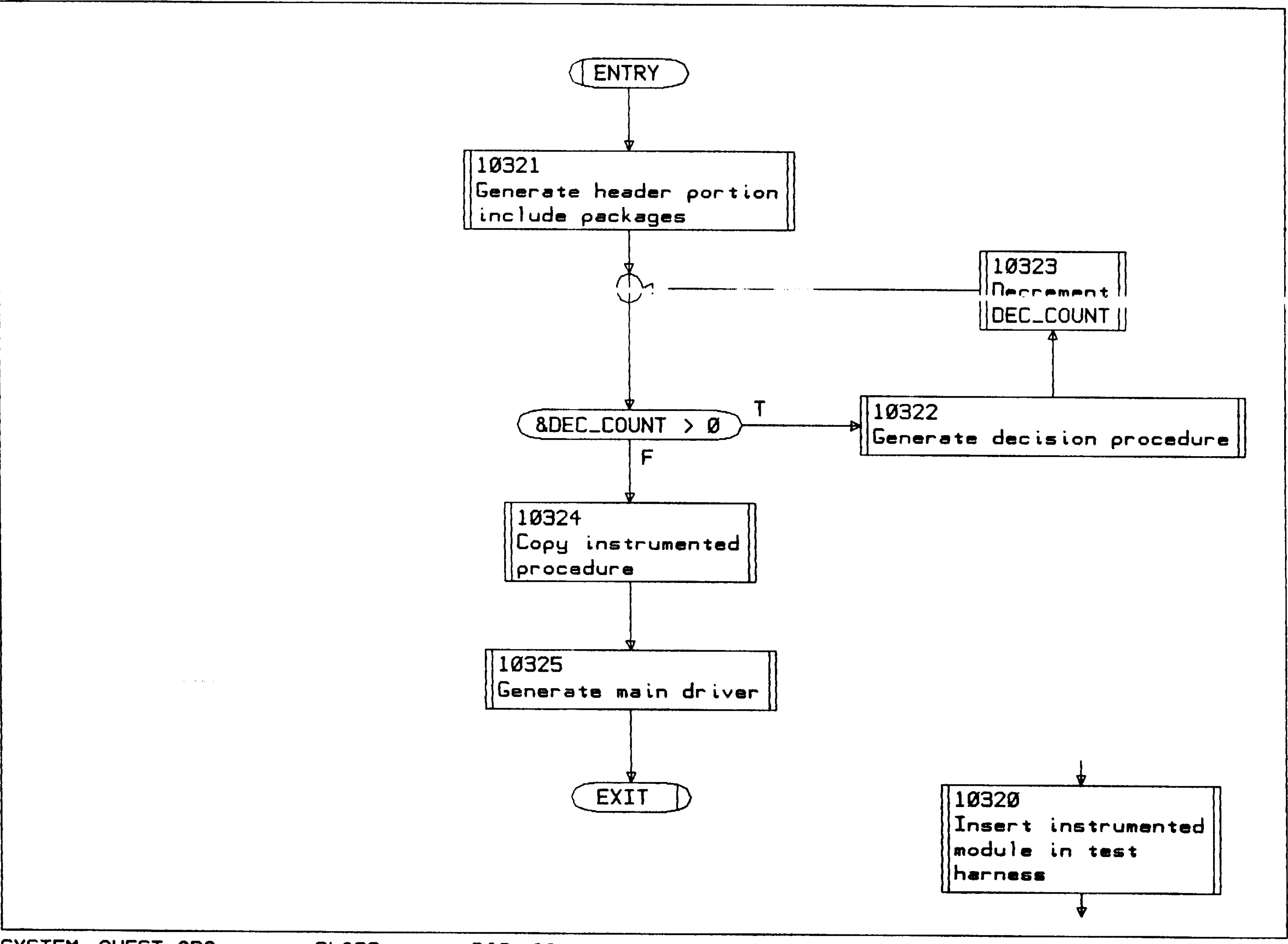
SEC: PPD-10200

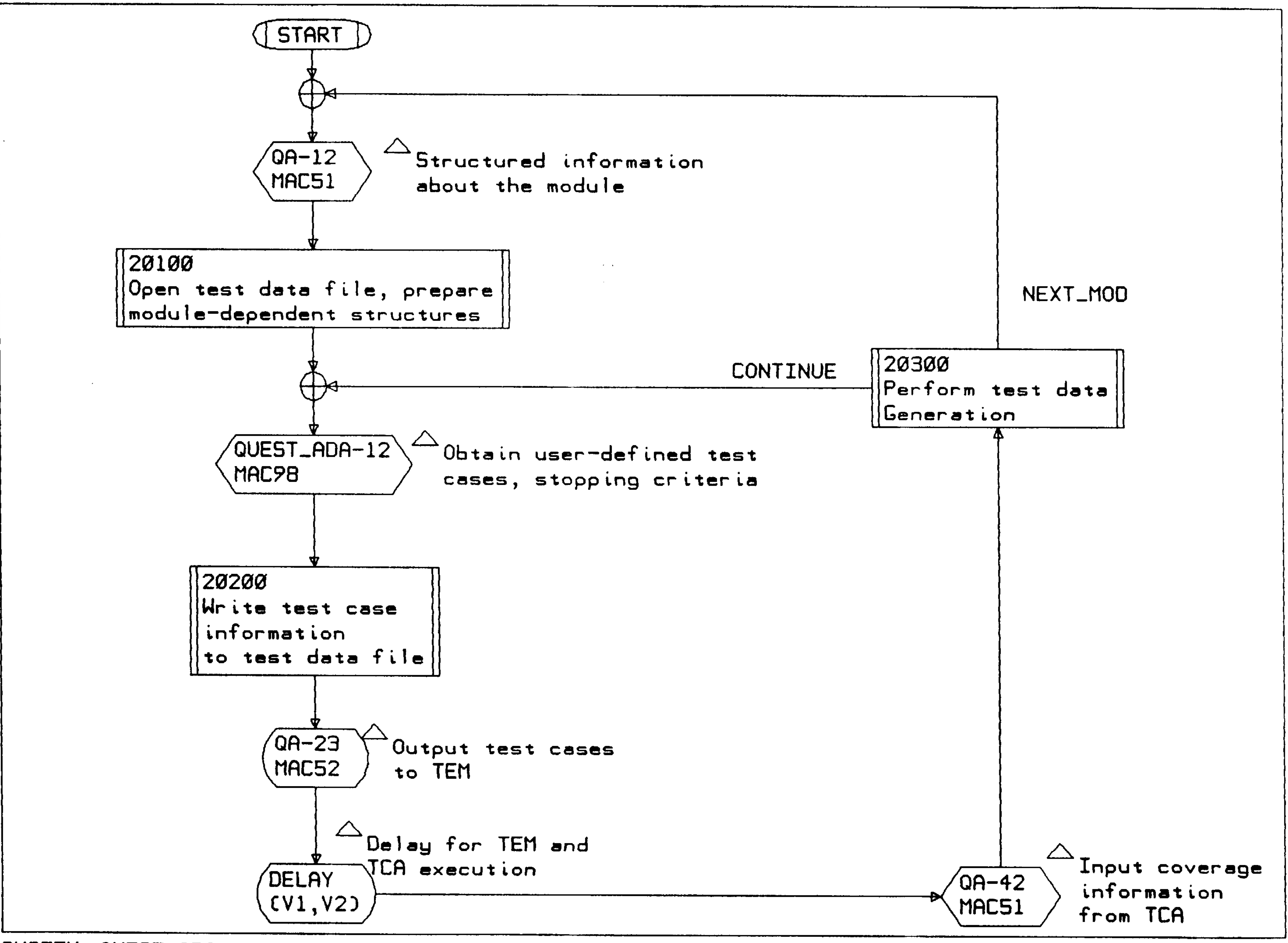
PAGE: 1
DATE: 5-Jul-89

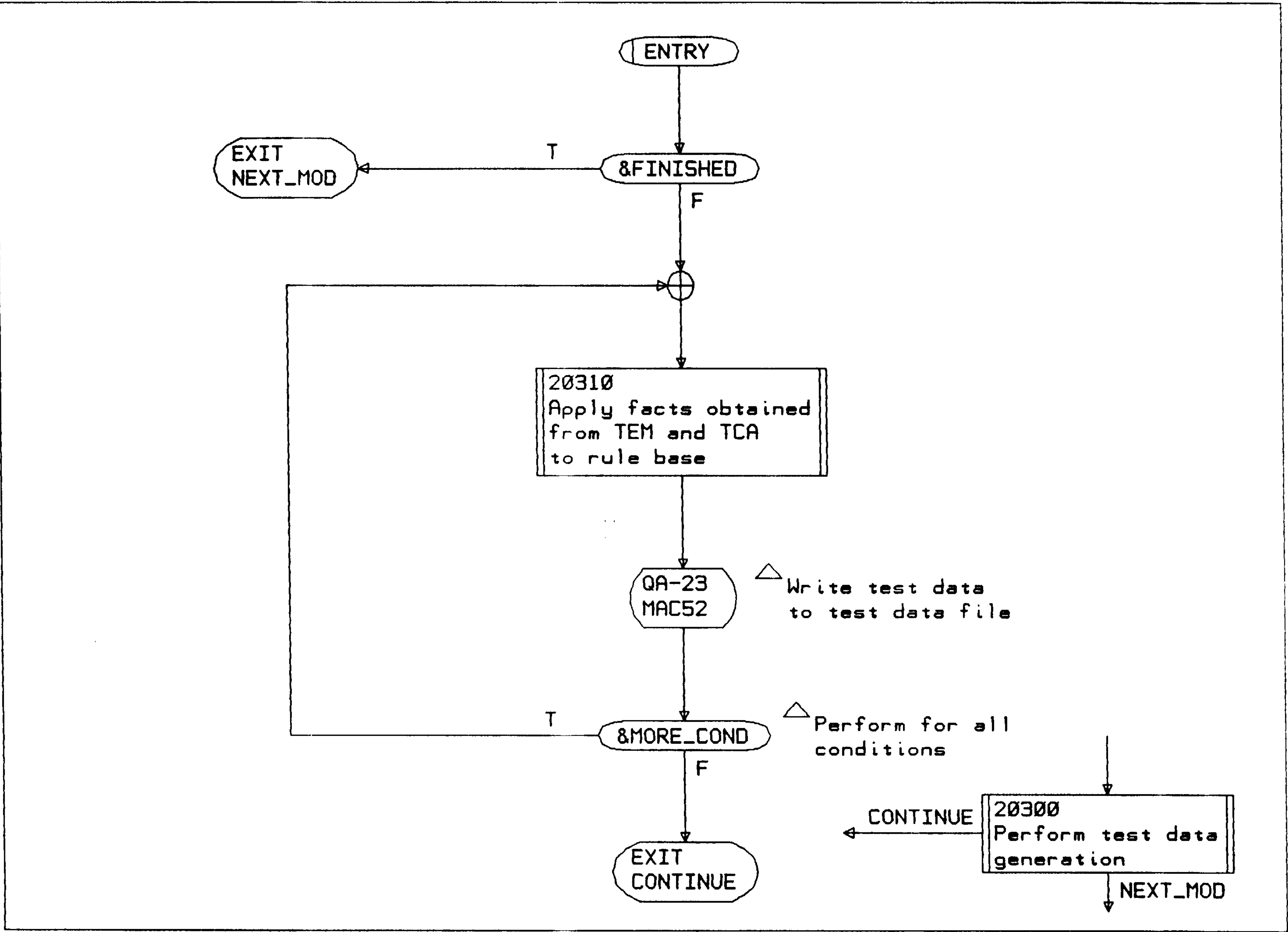


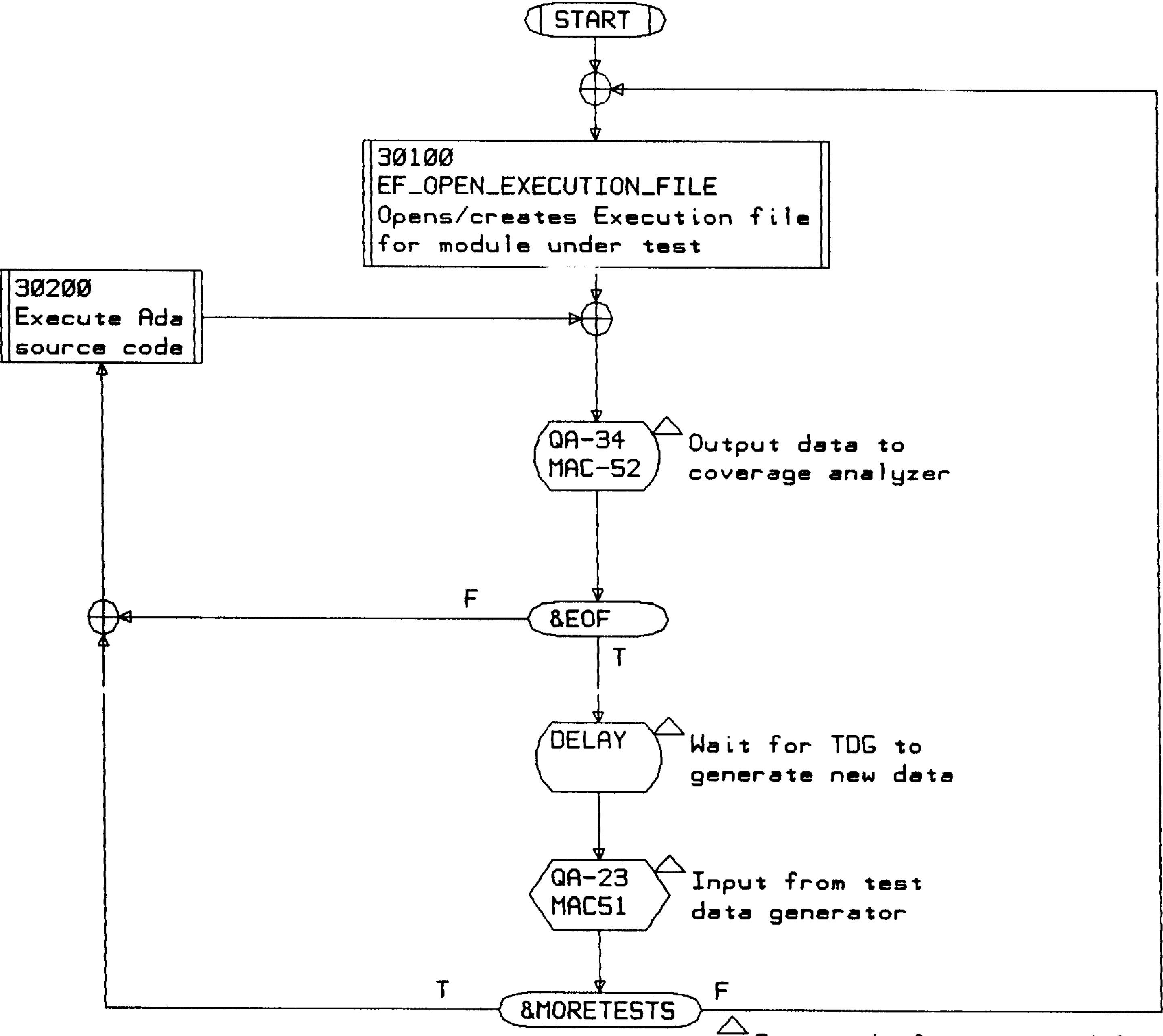


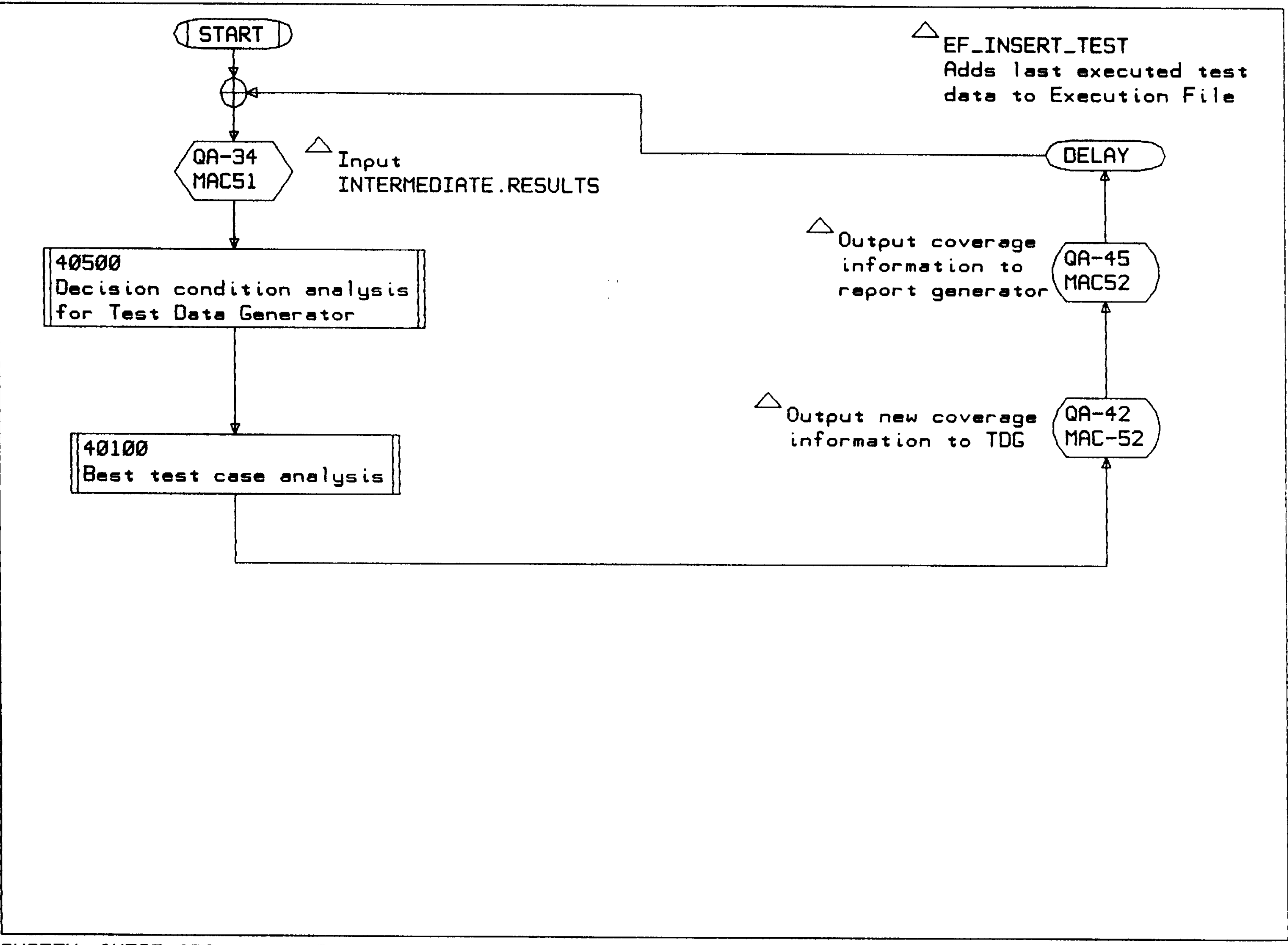
SYSTEM: QUEST_ADA CLASS: DOC: QA SEC: PPD-10300 PAGE: 1
MAP: DEFAULT_MAP ISSUE: D DATE: 12-Jun-89

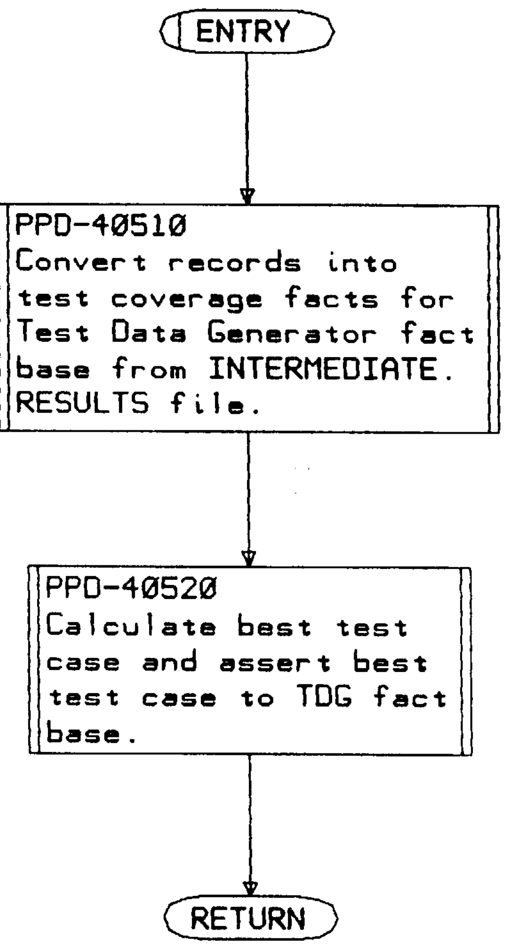

ENTRY
10321
Generate header portion
include packages
10323
Decrement
DEC_COUNT
&DEC_COUNT > Ø
T
F
10322
Generate decision procedure
10324
Copy instrumented
procedure
10325
Generate main driver
EXIT
10320
Insert instrumented
module in test
harness
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: PPD-10320
PAGE: 1
DATE: 12-Jun-89


START
QA-12
MAC51
Structured information
about the module
20100
Open test data file, prepare
module-dependent structures
QUEST_ADA-12
MAC98
Obtain user-defined test
cases, stopping criteria
20200
Write test case
information
to test data file
QA-23
MAC52
Output test cases
to TEM
Delay for TEM and
TCA execution
DELAY
(V1,V2)
QA-42
MAC51
Input coverage
information
from TCA
20300
Perform test data
Generation
CONTINUE
NEXT_MOD
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: IORTD-2
PAGE: 1
DATE: 12-Jun-89

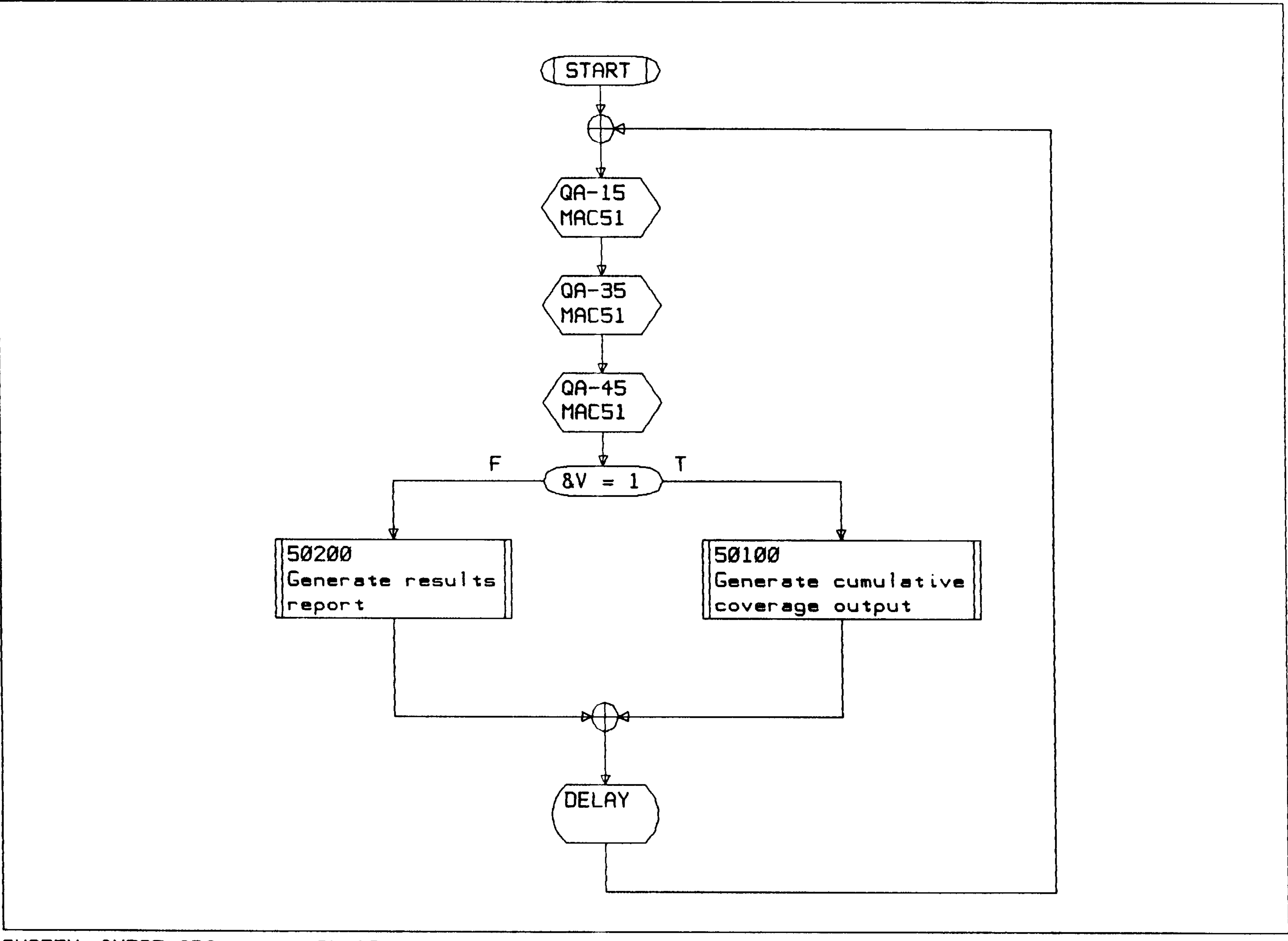

ENTRY
&FINISHED
T
F
EXIT
NEXT_MOD
20310
Apply facts obtained
from TEM and TCA
to rule base
QA-23
MAC52
Write test data
to test data file
&MORE_COND
T
F
Perform for all
conditions
EXIT
CONTINUE
20300
Perform test data
generation
CONTINUE
NEXT_MOD
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: PPD-20300
PAGE: 1
DATE: 12-Jun-89


START
30100
EF_OPEN_EXECUTION_FILE
Opens/creates Execution file
for module under test
30200
Execute Ada
source code
QA-34
MAC-52
Output data to
coverage analyzer
&EOF
F
T
DELAY
Wait for TDG to
generate new data
QA-23
MAC51
Input from test
data generator
&MORETESTS
T
F
Get ready for next module
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: IORTD-3
PAGE: 1
DATE: 12-Jun-89


START
QA-34
MAC51
Input
INTERMEDIATE.RESULTS
40500
Decision condition analysis
for Test Data Generator
40100
Best test case analysis
EF_INSERT_TEST
Adds last executed test
data to Execution File
DELAY
Output coverage
information to
report generator
QA-45
MAC52
Output new coverage
information to TDG
QA-42
MAC-52
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: IORTD-4
PAGE: 1
DATE: 12-Jun-89



SYSTEM: QUEST_ADA CLASS: DOC: QA SEC: PPD-40500 PAGE: 1
MAP: DEFAULT_MAP ISSUE: D DATE: 24-Feb-89


START
QA-15
MAC51
QA-35
MAC51
QA-45
MAC51
F
&V = 1
T
50200
Generate results
report
50100
Generate cumulative
coverage output
DELAY
SYSTEM: QUEST_ADA
MAP: DEFAULT_MAP
CLASS:
DOC: QA
ISSUE: D
SEC: IORTD-5
PAGE: 1
DATE: 12-Jun-89

APPENDIX B

A RULE-BASED SOFTWARE TEST DATA GENERATOR

The paper given in this appendix was produced in part by support provided by this project contract. This paper has been submitted for consideration for publication in IEEE Transactions on Knowledge and Data Engineering.

A Rule-Based Software Test Data Generator

by

William H. Deason

David B. Brown, Member, IEEE

Kai-Hsiung Chang, Member, IEEE

James H. Cross II, Member, IEEE

Address response to:

Dr. David B. Brown
Professor and Interim Head
Department of Computer Science and Engineering
107 Dunstan Hall
Auburn University, Alabama 36849-5347

(205) 844-4330

Submitted: November 15, 1988

Revised: August 15, 1989

ABSTRACT

Software reliability is of major concern in science and industry. Currently, software testing is the only practical means of assuring reliable software. To avoid the expensive manual tasks involved, software testing must be further automated to enable larger numbers of tests to be performed. A key component in an automatic software testing environment is the test data generator.

Rule-based software test data generation is proposed as an alternative to either path/predicate analysis or random data generation. A prototype rule-based test data generator for Ada programs was constructed and compared with a random test data generator. Four Ada procedures were used in the comparison. Approximately 2,000 rule-based test cases and 100,000 randomly-generated test cases were automatically generated and executed. The success of the two methods was compared using standard coverage metrics. Simple statistical tests were performed, which show that even the primitive rule-based test data generation prototype is significantly better than random data generation.

I. INTRODUCTION

*Software reliability** is one of the primary concerns of the computer science community and of scientific, commercial, and military organizations as well. Software testing is the only feasible means of assuring acceptable reliability for large software systems. However, test case development, execution, and evaluation are typically very time-consuming and labor-intensive tasks. In general, the tester must be satisfied with examining the results of a finite number of *test cases* and concluding that either (1) the reliability of the software is acceptable or (2) the software

---

* Italicized terms are defined in Appendix A.

contains *faults* which produce intolerable *errors*. In the former case, the software is installed for use, usually by being integrated into an overall system (with accompanying integration testing). In the latter case, additional resources must be applied for *debugging* and *regression testing* of the software. The alternative is either to use unacceptable software or to abandon the product development. Neither option is very inviting.

Fortunately, there is hope for improving this situation since much of the software testing process may be automated. Test execution may be accomplished by test drivers which are constructed by a software testing system. Test execution results may be automatically compared to outputs of a design-specification simulator or a redundant implementation of the software component. Test set adequacy may be monitored as a termination condition for the testing process. While these capabilities are not simple to achieve, they are relatively well understood. However, automated test data generation is not well understood [12, 15].

The approach typically taken has been to attempt to generate the least number of tests that will guarantee a certain level of test adequacy. This approach is applicable when test results must be manually validated against design specifications. However, it is insufficient for ensuring reliable mission-critical software. Orders of magnitude more tests are required, which are only feasible given the use of simulation or redundant coding for output verification. In this new scenario of very large test sets, test data generation techniques are needed which are able to generate large amounts of effective test data. One simple approach is to use a random number generator to generate the data. This is generally considered to be ineffective in that it rarely provides the necessary coverage of the program. This paper proposes a new rule-based approach to test data generation and then demonstrates that it can easily produce a large amount of test data which provides a much greater degree of coverage than randomly selected data. This new approach provides an extensible framework which utilizes newly developed heuristic rules as well as existing ones that have been used manually in traditional testing environments.

II. BACKGROUND

A. General

Software testing, as referenced in this paper, is strictly dynamic testing, which is the execution of programs with specific input data and the production and assessment of outputs [23]. This type of software validation takes place in the programming and maintenance phases of the software life cycle. It is recognized that verification and validation techniques must be employed also during the requirements definition and design specification phases, as the cost of eliminating bugs is higher the later they are uncovered in the software life cycle [6, 8]. A test case is a formally produced collection of prepared inputs, predicted outputs, and observed results of one execution of a program [1].

Oracles are external sources of information used to detect occurrences of errors. Oracles may be detailed requirement and design specifications, examples, or simply human knowledge of how a program should behave. An oracle is capable of determining whether or not a program has executed correctly on a given test case [9]. An oracle is required for dynamic testing of software function, and must be employed, either by testing personnel or by an automated testing system, to determine whether outputs are correct. Two automated forms of oracles, cited above, are design specification simulators and redundant manual code implementations.

A test adequacy criterion is needed to determine when to stop testing. Such a criterion is called program-based if it is based on the program code rather than an independent design specification of the program. *Statement coverage* and *branch coverage* are two program-based test adequacy criteria [23]. *Instrumentation* of programs aids in evaluating how well adequacy criteria have been met. Instrumentation is the insertion of additional statements into the program which, when the program is executed, will compute some dynamic attributes of the program [11]. For example, a simple instrumentation scheme could insert counters to record the number of times each statement is executed.

B. Test Data Generation

Test data generation has been defined as specifying and providing the test input data and calculating the test output data [20]. Generating test inputs for a program may not appear to be a difficult problem since it may be done by a random number generator [5]. However, random test data should not be expected to satisfy test adequacy criteria as well as selectively chosen test data. On the other hand, algorithms for generating test data to satisfy particular adequacy criteria have generally had very poor time and space complexities, thus producing small amounts of test data. In fact, it is not possible (that is, there exists no algorithm) to generate test data which causes the execution of an arbitrary program path [12]. This is the predicate solution problem, which reduces to the halting problem. For example, the path-predicate generation/solution approach for statement coverage must: (1) choose, from the (possibly infinite) set of possible paths through the program, a subset of these paths which will provide statement coverage, (2) construct a path predicate for each chosen path, and then (3) solve the associated path predicate for each path in terms of the inputs to the program.

Although the predicate solution problem is extremely complex, and no algorithm exists for solving general nonlinear predicates [12], there are some methods which will find solutions to many predicates. One implementation of the path predicate methodology is Query Utility Environment for Software Testing (QUEST) [2, 21, 22]. QUEST is applicable to a subset of FORTRAN 77 and provides options to attempt to generate test data to satisfy *statement coverage*, *decision coverage*, *condition coverage*, or *decision/condition coverage*. Of course, there is no guarantee that the predicate solution algorithm will be able to solve a given predicate; it must halt after a predefined number of unsuccessful attempts to find a solution and resort to some alternative such as random test case generator. Even for those predicates, which can be solved, each solution yields input data for only one test execution. This is a third problem with traditional test generation methods: they produce a relatively small number of (possibly trivial) test cases. The problem, then, is to propose and evaluate an alternative to either manual or predicate-solution test case generation methods.

DeMillo, Lipton, and Sayward [3] attempted to develop a practical test data generation methodology somewhere between random data generation and full program predicate solution. Noting that programmers produce code that is close to being correct, they observed the coupling effect property which is the ability of test cases, designed to detect simple errors, to surface more subtle errors as well. Howden, on the other hand, developed a set of functional testing rules [10]. Although both of these research efforts were directed at helping programmers test their code manually, they are also directly applicable to automatic test data generation. Instead of algorithms they are useful rules of thumb, often called heuristics, which embody certain bits of "expert knowledge." Thus, a knowledge-based or expert system approach is appropriate for attacking the problem of generating test data for software programs. Such an approach is made possible not only by the maturing body of knowledge about software testing, but also by developments in the field of rule-based systems, a branch of artificial intelligence. Both the coupling effect and Howden's functional testing rules are important to the rule-based approach presented in this paper.

III. THE RULE-BASED APPROACH

Since the manual rules of thumb or heuristic methods can be put in a rule base, the first step to full automation is the development and evaluation of such a rule base. The next step is the development of a parser/scanner mechanism to generate the information from the code itself to drive the rule base for automatic test case generation. The proposed paradigm not only draws information from the code itself, it also uses the results of prior tests as input for the generation of additional test cases. In this section, a brief overview of the rule-based approach is presented, followed by a detailed discussion of rule development criteria and a description of the ten rules that were developed and used in the prototype.

A. The Rule-Based Model

Figure 1 shows a data flow diagram of our rule-based test data generation system. In this model, test data generation rules (Rule Base file) and symbol information specific to the subject module under test (Symbol Table file) are used by the Rule Interpreter to generate test cases for the subject module. A Driver Program is used to control the execution of the subject module, which has been *instrumented* to collect coverage information with respect to the generated test cases. The execution results (Prolog-Readable Results file), consisting of the test cases and coverage statistics, are then used by the Rule Interpreter to generate additional test cases. This cycle continues until the amount coverage requested by the tester is achieved or until the requested number of test cases has been generated and executed. At this point, the Human-Readable results file is examined to determine the coverage achieved. The coverage metrics computed are *condition coverage*, *decision coverage*, and *multiple-condition coverage*. The items in Figure 1 will be addressed in more detail during the discussion of the prototype.

B. Rule Development Criteria

Before developing a rule base for test data generation, a test adequacy criterion must be established to provide the goal for rule development. Several different criteria were evaluated, including *statement coverage*, *condition coverage*, *decision coverage*, and *multiple-condition coverage*, and a selection was made based upon the strength of the adequacy criterion. The strength of a criterion generally reflects the number of tests required to satisfy that criterion. Assuming that the outputs of all test cases are checked for functional correctness, the satisfaction of stronger criteria also provides more evidence of the correctness of the program under test. For these reasons, *condition coverage*, *decision coverage*, and *multiple-condition coverage* were selected as criteria for the prototype. Thus, the rules which were developed attempted to define a process by which the test cases generated would satisfy these criteria.

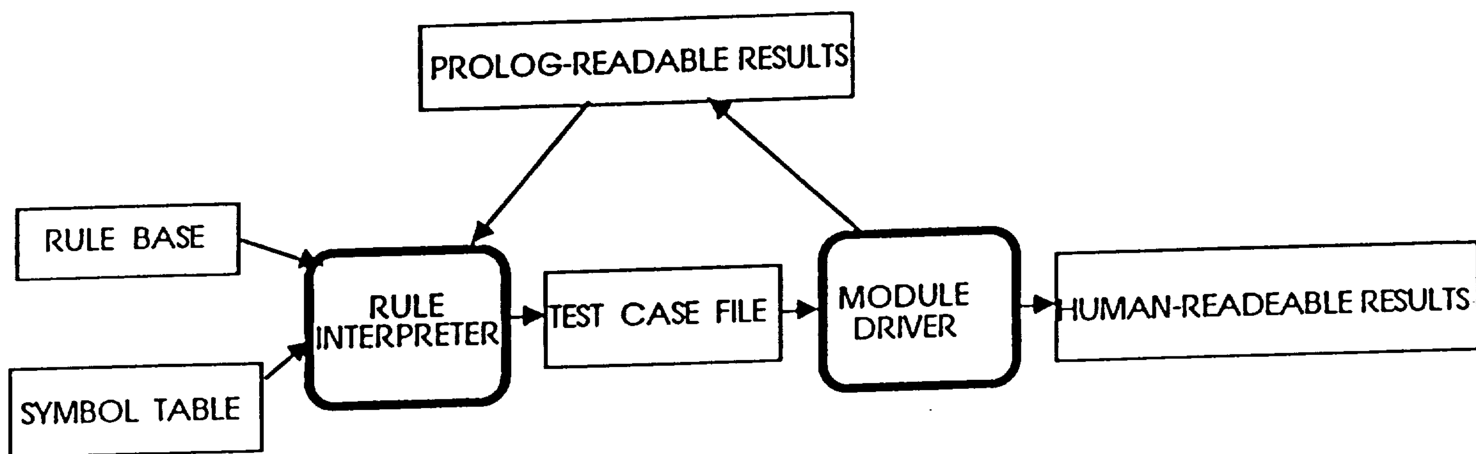


Figure 1. Rule Based TCG Paradigm

A test data generation rule, which is represented as a Prolog clause, consists of two parts: the IF part (or preconditions), and the THEN part (or actions) of the rule. The IF parts of the rules are typically their physical requirements, reflecting the fact that a rule could possibly be applied. The THEN parts of the rules consist of action statements which create test cases for future execution. Examples of rules written in Prolog are given in the next section.

Before the rules can be defined, the relative value or merit of individual test cases must be understood. The rule-based test data generator is designed to function in an iterative manner. One iteration consists of: 1) generating new test cases based on previously executed test cases, 2) executing the new test cases, and 3) updating the cumulative execution results. This execution information consists of the two "best" test cases executed to that point for each condition. Only these two test cases (i.e., one for the true and one for the false outcome) are used as a basis for the next iteration of test data generation rules. If the number of test cases saved from iteration to iteration was not limited, the search process would be an exhaustive breadth-first search, the number of test cases generated per iteration would be very large, and the entire process would be rendered ineffective. The iterative procedure used the concept of test case "goodness", which requires more precise definition. A test case T1 will be considered better than another test case T2, with respect to the condition C1, if: (1) C1 is a relational expression of the form

LHS <relop> RHS

where <relop> is any relational operator, LHS is the left hand, and RHS is the right hand side of the relation; and (2) the percent difference between the values taken on by LHS and RHS during a given test case, T1, is less than the percent difference between the values of LHS and RHS during test a succeeding test case, T2. The percent difference between LHS and RHS is defined as:

$$ABS(LHS - RHS) / MAX(LHS, RHS)$$

The terms LHS and RHS in the percent difference formula represent the values that LHS and RHS take on during a particular test case execution. The entire test data generation process may be viewed as an attempt (guided by rules) to minimize the percent difference between the values of LHS and RHS of each condition in the module under test. This definition of test case "goodness"

holds because it is generally true that test cases closer to condition boundaries are superior in that they provide more information about the correctness of the conditions. Also, in a case where one of the two outcomes of a condition has not been executed at all, test cases closer to the boundaries are usually more likely to lead to a test case which crosses the boundary and covers the opposite outcome.

The rationale for rule development given above is proposed merely to provide a starting point for rule development. Recognize that the objective here was not to develop the ultimate rule base. Rather it was to test the concept of rule-based test case generation in order to validate the design paradigm which will be described below. With these preliminary definitions in mind, we can now proceed to describe the set of rules used in the evaluation.

C. Rules

This section describes a trial set of rules developed to generate test data. A narrative is given for each rule describing its rationale and explaining implementational details as necessary. As discussed earlier, most of these rules are based on the ideas developed by DeMillo, Lipton, and Sayward [3] and Howden [9], who are considered to be the experts in heuristically generated software test data.

In the following discussion, a test case is considered to be a list of values, (v1, v2, ... , vn). Each value corresponds to an input variable of the procedure to be tested. Since a condition may not involve all input variables, the best test case for each condition will generally differ from the others. Suppose a condition, COND, involves only the ith variable. Its best test case (v1, v2, .. ,vi,.., vn) would force the execution of COND while providing the smallest percent difference. If a further improvement is required with respect to COND, only the value of the ith variable will be modified.

The rule base contains 10 rules. Each rule is capable of generating multiple test cases. In each iteration, the rules are scanned one by one. Whenever a rule is applicable (or its IF-part is satisfied), its test case generation action is taken. Most of the time, one iteration will "fire" more than one rule, thus generating multiple test cases for a condition.

Rule 0:

IF: None (always applicable)

THEN: Generate tests with random values for each of the inputparameters.

Rule 0 provides the starting values for test data generation. When the automatic test data generator is used to test code, these starting points should not be random; rather, they should be provided by the designer or the tester of the program. In fact, an entire suite of predesigned test cases could be substituted for this rule in order to initiate testing. However, the existence of such human-provided test cases will not be assumed. Since this would unfairly bias our evaluation, which compares the rule-based test cases against random test cases. Rule 0 generates three test cases, with values in the range -1 ... +1, -100 ... +100, and -1000 ... +1000. A slight variant of this rule could take advantage of subtype ranges by picking R for a particular subtype based on the actual range of the subtype. Unlike the rest of the rules, this rule does not require any previously executed test cases. A simplified Prolog version of this rule is shown below.

```
rule(1):- in_parms(INPARMS),                ;number of parameters
          random_tests(INPARMS,1),          ;first case, range -1 to 1
          random_tests(INPARMS,100),        ;second case
          random_tests(INPARMS,1000).       ;third case

random_tests(ARGC,MAX):-                    ;random number generator
          one_random_test(ARGC,MAX,NEWARGS,1),
                                            ;one new case in NEWARGS
          save_a_test(NEWARGS).             ;save a test case
```

Rule 1:

IF: The program contains a condition which contains an input variable and a constant, and the best test so far for a (True or False) outcome of the condition gave a percent difference greater than 5%.

THEN: Generate a test case from the previous best test case by putting the value of the constant in the position of the input variable contained in the condition.

According to the criterion given in the previous section, Rule 1 is designed to test conditional expressions of the form

X <relop> K

where X is an input parameter, K is a constant, and <relop> is any relational operator. This rule comes directly from the handling of arithmetic relations in Howden [9]. However, the reason this rule is applied to more complex expressions is that it may provide good tests because of the coupling effect. It may also provide a good approximation which may be refined to achieve better testing of these expressions. A simplified Prolog version of this rule is shown below.

```
rule(1):- rule1_applies(C,V,P,K,OLDTC), ;precondition checking
          newtest(OLDTC,P,K,ARGS),      ;replace the value of
                                        ;variable P in OLDTC by
                                        ;K, result is ARGS
          save_a_test(ARGS).            ;save a test case

rule1_applies(C,V,P,K,OLDTC):-          ;preconditions for rule1
          testfor(OLDTC,C,VAL,OLDCOV),  ;there is a case, OLDTC,
                                        ;for condition C. VAL is
                                        ;not used here.
          OLDCOV > 5,                   ;percent difference > 5%
          condition(C),                 ;C is a condition
          variable(C,V),                ;C's variable list is V
          in_parm(V,P),                 ;variable P is in V
          constant(C,K).                ;K is a constant in C.
```

Rule 2:

IF: The program contains a condition which contains an input variable and two constants, and the best test so far for a (True or False) outcome of the condition gave a percent difference greater than 5%.

THEN: Generate three test cases from the previous best test case by putting the sum, then the differences, of the constants in the position of the input variable contained in the condition.

Rule 2 is designed to test expressions of the form:

X + K1 <relop> K2

or

X - K1 <relop> K2

where K1 and K2 are constants. Solving each of these equations for X yields the expressions K2-K1 and K1+K2. Therefore, K1+K2, K1-K2, and K2-K1 are values used by rule 2.

Rule 3:

IF: The program contains a condition which contains an input variable and a constant, and the previous best test for a (True or False) outcome of the condition gave a percent difference greater than 5%.

THEN: Generate two test cases from the previous best test case by putting a value slightly greater than the constant, then slightly less than the constant, in the position of the input parameter contained in the condition.

Rule 3 is designed to cover conditional expressions of the form

X <relop> K

where X is an input parameter and K is a constant. While rule 1 generates an "on" point for these types of conditions, rule 3 generates two "off" points, that is, slightly off the subdomain boundary formed by the conditional expression. As with rule 1, rule 3 comes directly from the handling of arithmetic relations [9].

Rule 4:

IF: The program contains a condition which contains an input variable and two constants, and the best test so far for a (True or False) outcome of the condition gave a percent difference greater than 5%.

THEN: Generate three test cases from the previous best test case by putting the product of the constants, then the ratio of the constants, in the position of the input variable contained in the condition.

Rule 4 is designed to cover expressions of the form:

X * K1 <relop> K2

or a similar form. It uses K1*K2, K1/K2, and K2/K1 in order to cover these expressions.

Rule 5:

IF: The program contains a condition which contains an input variable and three constants, and the best test so far for a (True or False) outcome of the condition gave a percent difference greater than 5%.

THEN: Generate test cases from the previous best test case by putting the sum of two of the constants divided by the third, then the difference of two of the constants divided by the third, in the position of the input parameter contained in the condition.

Rule 5 is designed to test conditions of the form

K1 * X + K2 > K3

or similar forms. All possible combinations of K1, K2, and K3 are used so that the following values are computed:

(K1 + K2) / K3
(K1 - K2) / K3
(K2 - K1) / K3
(K1 + K3) / K2
(K1 - K3) / K2
(K3 - K1) / K2
(K2 + K3) / K1
(K2 - K3) / K1
(K3 - K2) / K1

Rule 6:

IF: An outcome of a condition has not been executed, there is at least one previously executed test case, and the procedure contains at least one constant.

THEN: Generate a test case from the previously executed test case by replacing an input variable with the constant.

Rule 6 was designed to use program constants to search for test cases to cover condition outcomes which have not yet been covered at all. However, Rule 6 proved to be inefficient and so was removed from the active rule base during the prototype evaluation phase of the project.

Rule 7:

IF: There is a test case which produces an outcome of a condition.

THEN: Generate test cases by incrementing and decrementing the values of the previous best test case.

Rule 7 is the first of the purely search-oriented rules. It varies, by a small amount, the input variable values in the best test case for an outcome of a condition. It is primarily intended to improve the coverage of a condition outcome, although it may in some cases cause the opposite outcome to be executed. The latter is desirable when the opposite outcome has not been covered by any previously executed test case. This general approach was used quite successfully by Prather [16].

Rule 8:

IF: There is a test case for an outcome of a condition.

THEN: Generate test cases by doubling and halving the values of the previous best test case.

Rule 8, like Rule 7, is a purely search-oriented rule. Rather than changing the values by a small amount, as Rule 7 did, Rule 8 varies the values by doubling and halving them. While Rule 8 certainly provides much less precision than Rule 7, it allows much faster movement through the search space.

Rule 9:

IF: There is a test case for an outcome of a condition.

THEN: Generate test cases by replacing a value in the test case with a random number.

Rule 9 is a partially random search rule in that it randomly changes one of the inputs in the test case while holding the other inputs constant. This rule may cover conditions of the program when the other rules fail.

IV. PROTOTYPE IMPLEMENTATION

After developing a speculative set of test data generation rules, it was necessary to implement a prototype test data generator employing these rules in order to evaluate the ability of a rule-based test data generator to produce good test cases. The prototype, which was implemented in Prolog, is designed to generate test cases and analyze coverage for a subset of Ada.* For the purposes of the prototype, each test case was simply a set of input values (i.e., expected results were not required to demonstrate coverage). In this section, the prototype support for the subset of VAX Ada is described, followed by a discussion of the rule interpretation, test execution, and coverage evaluation portions of the prototype.

A. Prototype Support For Ada Subset

The scope of the prototype implementation was limited in two major ways. First, only subprogram input parameters were considered as inputs to the subprogram under test. That is, no files were generated to test programs which process files. Second, the type of inputs allowed was limited to the VAX Ada types INTEGER and FLOAT, defined in the package STANDARD. The INTEGER type was chosen to represent all discrete types, such as enumerated types, in that these types map to a subset of the integers. The FLOAT type is representative of real number types. Thus, the application of rule-based test data generation to these two data types will demonstrate its applicability to most numeric types, and will provide some evidence of its applicability to more complex types. While these limitations must be relaxed when this approach is actually applied in practice, they are no hindrance to demonstrating the potential value of rule-based test case generation.

The semantic information required by the expert test data generator is not nearly as detailed as that required by a compiler. It could easily be output as a by-product of the compilation of Ada code. The description of a program to the rule-based test data generator must contain: 1) the

---

* Ada is a trademark of the United States Government, Ada Joint Programs office.

names and types of input parameters, 2) the conditions of the program, and 3) the variables and constants contained in these conditions. Since the test data generator expert system prototype is implemented in Prolog, the information must be provided in the form of Prolog facts. In the present prototype, the SYMBOL TABLE for each subject module under test was hand-coded. In future versions of the prototype, a specialized parser/scanner will be used to generate these Prolog facts directly from Ada source code.

B. Rule Interpreter

The Rule Interpreter, which controls the entire testing process, was written in Prolog to expedite the implementation of the prototype. As seen in Figure 1, the Rule Interpreter reads in the information about a subject module from the Symbol Table file, uses the Rule Base to generate test data, and then calls a Driver Program to execute the subject module using these test cases. In particular, once the Prolog interpreter is activated, it queries the user for the name of the subject module to be tested, the number of iterations, and the maximum number of test cases to be generated during a single iteration. The next step causes all applicable rules (Prolog clauses) in the Rule Base to fire using the subject module Symbol Table information (Prolog facts) and, if available, previous execution results. The test cases generated by the rules are placed in the Test Case File, and control is passed to the Driver Program, which executes the subject module and records the input values and coverage matrix for of each test case in the Execution Results File. When control returns to the Rule Interpreter, the success of each test case is evaluated based on the execution results. The last action is to succeed (stop) if the desired iterations have been performed; otherwise the Rule Interpreter continues the testing process by recursively calling itself.

C. Module Drivers and Instrumentation

Each iteration of the Prolog rule interpreter may generate many test cases. These test cases are stored in the Test Case File. For this reason, each procedure being tested must have a Module

Driver, that is, a program which reads the test file, executes the procedure, and records the results in the two results files. This process is repeated once for every test case in the test case file. The Module Driver, which is written in Ada, consists of two parts: 1) the procedure being tested and 2) the instrumentation procedures, which measure coverage. The driver algorithm (in pseudocode) is as follows:

```
repeat for all tests in test case file
        initialize coverage matrix
        execute procedure under test with test case
        output input values and coverage results
```

The instrumentation procedures are all named CONDITION, which is allowed by Ada overloading. This fact makes the instrumentation easier than it otherwise might be. Two different forms of the CONDITION procedure are used. The simplest is used to instrument conditions which do not contain a relational operator, such as Boolean function calls. For instance, suppose there is a function which returns the type BOOLEAN (true or false) and whose value simply indicates whether or not its one integer argument is a prime number. A statement such as this might appear:

```
if IS_PRIME(I) then...
```

This statement would be instrumented as follows, assuming that this is the third condition in the program:

```
if CONDITION(3,IS_PRIME(I)) then...
```

The action of this form of CONDITION is simply to note in the coverage matrix whether condition number three executed true or false (the value returned by IS_PRIME). Then, CONDITION returns the same BOOLEAN value that IS_PRIME returned to it, so that the program continues to execute as it would have without the instrumentation.

The second form of the CONDITION procedure is slightly more complicated. It is used to instrument conditions of the form

<expression> <relop> <expression>

such as X>2, X*Y<Z, and X**2+Y**2=Z**2. This form of the CONDITION procedure takes four arguments: 1) the number of the condition, 2) the expression to the left of the relational operator, 3) an enumerated-type value indicating the relational operator, and 4) the expression to the right of the relational operator. The three previous example expressions would be instrumented as follows, assuming that they are the first three conditions in the procedure under test:

```
CONDITION(1,X,GT,2)
CONDITION(2,X*Y,LT,Z)
CONDITION(3,X**2+Y**2,EQ,Z**2).
```

In summary, module drivers and instrumentation were required in order to evaluate the prototype rule-based test data generator. Their function was the same as that required for traditional testing methods: to facilitate test case execution and to evaluate coverage, respectively. A specialized parser/scanner is being constructed for the purpose of generating the instrumentation/driving mechanism as well as the Prolog facts (Symbol Table File) that describe the subject module under test. These functions are candidates for future inclusion in commercial Ada compilers.

V. EVALUATION OF PROTOTYPE

After developing the prototype test data generator, it was necessary to design a formal procedure for evaluating the prototype. The test data produced by the prototype was compared, using the test adequacy criteria described earlier, with randomly generated test data. Four Ada procedures (TRIANGLE, ITRIANGLE, CURVE, LINEAR) were used to evaluate the test data generator. Although the procedures are small, each contains fairly complex conditional

expressions on its branch statements, and relatively complicated combinations of branch statements. Most of the path predicates for each of these procedures would be complex and quite difficult for automatic solution using predicate solution techniques. Each procedure is briefly described below.

The Ada procedure TRIANGLE accepts three inputs, each of the Ada type FLOAT. It returns a value of type INTEGER indicating which of several types of triangle is formed by taking the first two arguments as the two legs of a triangle, and the third argument as the hypotenuse.

The Ada procedure ITRIANGLE accepts three inputs, each of the Ada type INTEGER. Otherwise, it performs the same function as TRIANGLE, which receives inputs of type FLOAT. ITRIANGLE returns a value of type INTEGER indicating which of several types of triangle is formed by taking the first two arguments as the two legs of a triangle, and the third argument as the hypotenuse.

The Ada procedure CURVE accepts four inputs, each of the Ada type FLOAT. These four inputs represent the X and Y coordinates of two points in two-dimensional space. CURVE returns a value of type INTEGER indicating which of several types of curve best fits these two points. For example, the test case (1,1,2,2) would represent the points (1,1) and (2,2), and CURVE would return a value indicating that these points roughly fit an upwardly-sloping diagonal line.

The Ada procedure LINEAR accepts three inputs, one of the Ada type FLOAT and two of the Ada type INTEGER. The procedure is called LINEAR because it is composed of all linear conditional expressions. It performs no useful function.

Table 1 each of the procedures in terms of number of inputs, conditions, decisions, paths, and calls to subprocedures. For example, procedure TRIANGLE requires 3 input values and contains 13 conditions, 10 decisions (based on the 13 conditions), and 28 paths defined by the 10 decisions. In addition, TRIANGLE makes a call to one procedure. These item provide a basis for test adequacy criteria.

Table 2 presents a comparison of the coverage achieved for each procedure by the two methods of test data generation: the prototype test data generator ("Rules") and a random test data generator ("Random"). Each row of the table represents a single *test suite* for the indicated procedure and method of test generation. For example, the first row indicates that for procedure TRIANGLE, the prototype was used to generate 45 test cases which, when executed, covered 20 of 26* possible condition outcomes, 14 of 20 possible decision outcomes, and 18 of 26 possible multiple condition outcomes. Complete coverage is achieved whenever all possible outcomes (true and false) are invoked. However, this can never be guaranteed for an arbitrary program since it may contain *infeasible paths*.

In all, five test suites of 45, 155, 308, 429, and 504 test cases respectively were generated by each method. Of the 15 different combinations of five test suites and 3 standard coverage metrics for TRIANGLE, the prototype-generated test data obtained better coverage than the random test data nine times, and the random test data obtained better coverage five times. In the remaining case the coverage was the same. A chi-squared test was performed in order to test the statistical significance of the number of times the rule-based data outperformed the random data. The chi-squared value did not indicate a significant difference. However, if the first test suite (of only 45 tests) is neglected, then the rule-based data performs better nine of the twelve times and the random data performs better twice. The chi-squared value for this subset showed a significant difference with 95% confidence.

---

* The 26 possible condition outcomes for TRIANGLE are a result of its 13 conditions taking on values TRUE and FALSE.

Table 1. Procedures Used in Prototype Evaluation

	TRIANGLE	ITRIANGLE	CURVE	LINEAR
Inputs	3	3	4	3
Conditions	13	12	16	11
Decisions	10	9	13	8
Paths	28	28	9	9
Subprocedures	1	0	4	0

Table 2. Comparison of Rule-based with Random Data for the Four Ada Programs

PROCEDURE Method Used	Test Cases	Condition Outcomes Covered	Decision Outcomes Covered	Multiple-Condition Outcomes Covered
TRIANGLE		(of 26)	(of 20)	(of 26)
Rules	45	20	14	18
	155	21	15	19
	308	25	19	23
	429	25	19	23
	504	25	19	23
Random	45	22	15	20
	155	22	15	21
	308	22	15	21
	429	22	15	21
	504	22	15	21
ITRANGLE		(of 24)	(of 18)	(of 24)
Rules	49	21	15	18
	139	23	17	21
	270	24	18	22
	392	24	18	22
	461	24	18	22
	520	24	18	22
Random	49	21	11	19
	139	21	14	19
	270	21	14	19
	392	21	14	19
	461	21	14	19
	520	21	14	19

Table 2 Comparison of Rule-based with Random Data
for the Four Ada Programs (continued)

PROCEDURE Method Used	Test Cases	Condition Outcomes Covered	Decision Outcomes Covered	Multiple-Condition Outcomes Covered
CURVE		(of 32)	(of 26)	(of 32)
Rules	42	24	18	21
	94	28	22	25
	174	28	22	25
	188	28	22	25
	312	28	23	27
Random	42	15	12	12
	94	15	12	12
	174	15	12	12
	188	15	12	12
	312	15	12	12
LINEAR		(of 22)	(of 16)	(of 22)
Rules	73	13	8	11
	210	18	12	17
	321	18	12	17
	389	18	12	17
	428	18	12	17
Random	73	13	9	11
	210	13	9	11
	321	13	9	11
	389	13	9	11
	428	13	9	11

In an attempt to further discover differences in performance characteristics between rule-based and random data, more random tests were run on TRIANGLE to determine the number of random tests necessary to obtain the coverage obtained by the rule-based data. The random data covered 23 conditions after 640 tests, but attained no further coverage, even though 40,000 tests were run. This left the random data coverage still two conditions short of the coverage provided by the rule-based data.

A comparison of the coverage of ITRIANGLE achieved by the prototype test generator and a random test data generator for ITRIANGLE is shown next in Table 2. Of the 18 different combinations of six test suites and 3 coverage metrics, the prototype-generated test data obtained better coverage than the random test data 16 times, and the random test data obtained better coverage one time. In the remaining case the coverage was the same. This is obviously a highly significant difference (alpha < 0.005). As with the TRIANGLE procedure, additional random tests were performed. The random test data covered one more condition at test case 2216, and another at 7170, for a total of 23 conditions covered. This is still one condition short of the 24 condition outcomes covered by the rule-based data. A total of 20,000 random tests were performed for the procedure ITRIANGLE.

An interesting feature of the test data generation for the procedure CURVE is that the randomly generated data never improved over the initial random data. Even more importantly, the rule-generated test data obtained better values for all coverage metrics and for all test set sizes than the randomly-generated test data. Even at only 42 tests, condition coverage for the rule-based data was 60% better than the random, decision coverage was 50% better than random, and multiple-condition coverage was 75% better. When additional random tests were run for CURVE, three more condition outcomes were covered with 730 test cases, then two more with 1662 test cases, then one more with 1682 test cases. No more were covered up to 20,000 test cases. Cumulatively, 21 conditions were covered, which is seven short of the 28 conditions covered by the rule-based data.

Finally, a comparison of the coverage of LINEAR showed that in only one of the 15 standard coverage cases did the randomly generated data perform better than the rule-generated data. Only two cases was their performance the same. Chi-squared tests again showed a very significant difference (alpha < 0.005).

Additional random tests for LINEAR resulted in one condition outcome being added to the coverage for each of test case numbers 596, 1098, 1304, and 1778. The total conditions covered up to 20,000 test cases was 17, which is still one short of the 18 covered by the rule-based data.

DISCUSSION

While the primary objective of this work was to test the concept of rule-based test data generation, it also surfaced considerable knowledge on ways in which the rules can be further improved. For example, rules can be generated to simplify the expressions appearing in the conditions. Consider a condition, COND, having the format of: <exp1> <rel> <exp2>. By using the following simplification rules, the condition boundary of COND can be identified easier, and less test data needs to be generated to obtain the equivalent coverage:

Rule A

If <exp1> does not contain variables
then exchange positions of <exp1> and <exp2>

Rule B

If <exp1> contains constants
then move all possible constants to <exp2>

These rules would simplify <exp1> such that it contains at least one variable and no constants. For example, given a condition

3 =< 5 * X + 4

<exp1>: 3
<exp2>: 5 * X + 4
<rel> : =<

By applying Rule A, it becomes

5 * X + 4 >= 3

By applying Rule B, it becomes

X >= -0.2

From this, three test cases can be generated for X. They are X = -0.2 + e, X = -0.2, and X = -0.2 - e, where e is a relatively small number. Comparing with Rule 5 mentioned earlier, the original 9 test cases are reduced to 3 test cases with this simplification.

The following forms of expression are subject to Rules A and B:

		Example
1.	constant.	<exp> = 10
2.	single variable.	<exp> = x
3.	single variable + (-) constant.	<exp> = x + (-) 5
4.	single variable *(/) constant.	<exp> = x * (/) 5
5.	two variables (+,-).	<exp> = x + (-) y
6.	two variables (*,/).	<exp> = x * (/) y
7.	two variables + (-) constant.	<exp> = x +(-) y +(-) 5
8.	two variables * (/) constant.	<exp> = (x+(-)y)/5, or (x+(-)y)*5

Although there are 64 combinations between <exp1> and <exp2>, after simple simplification steps the combinations can be generalized into the following 10 cases.

	<exp1>	<exp2>
1.	X	C1
2.	X	Y
3.	X	Y + C1
4.	X	Y * C1 (or Y / C1)
5.	X	C1 * X + C2 * Y + C3
6.	X	C1 * X * Y + C2
7.	C1 * X + C2 * Y	C3 * X + C4 * Y + C5
8.	C1 * X + C2 * Y	C3 * X * Y + C4
9.	X * Y	C1
10.	X	C1 * Y / X + C2

As a further example, consider the sixth relationship given above. Since the goal of test data generation is to assure the generated test data will have small percent difference and cover both sides of the condition boundary, the place where a particular test case locates on the boundary is not critical. Thus we can determine Y as follows:

```
If    there is a best test case for this condition
then  assign Y = the value of Y in the best test case
else  assign Y = (upper-bound - lower-bound) of Y/2
```


C - 2

The test case value of X can then be determined by the following simplification steps.

<exp1>	<exp2>
X	C1 * Y * X + C2

Since the value of Y is now known, the relationship becomes

<exp1>	<exp2>
X	C3 * X + C2

By recursively applying Rule A and Rule B, we obtain the following:

<exp1>	<exp2>
X	C3 * X + C2
(1-C3) * X	C2
X	C4

From this relationship, the test case data is defined as:

1.	X = C4 + e	Y
2.	X = C4	Y
3.	X = C4 - e	Y

By using this type of simplification heuristics, more efficient test cases can be generated, i.e., fewer cases which cover more branches. It is expected that experience in exercising the rule base will lead to the generation of many other rules which will be subjected to comparative evaluation as the system is developed.

VI. CONCLUSIONS

The primary objective of this paper was to propose a rule-based software test data generator and to demonstrate its feasibility. This new approach to test data generation provides an

extensible framework which utilizes new heuristic rules as well as those which have been used manually in traditional testing environments. Such a test data generator would be used in conjunction with a software testing environment. The most important phases of the project were: 1) the development of a simple trial rule base, 2) the implementation of the prototype test data generator, and 3) the evaluation of the prototype. Ten test data generation rules were developed during the initial phase. During the second phase, these rules, along with a rule interpreter, were implemented in Prolog. Four Ada modules were selected and instrumented as test modules, and drivers were implemented for these modules. During the evaluation phase, approximately 2,000 rule-generated tests and 102,000 randomly-generated tests were executed in all. These two sets of data were compared using simple statistical tests. These tests clearly show that the rule-base-generated data is significantly better than the randomly-generated data. In fact, the same coverage could not be attained by random test-case generation even when very large numbers of randomly-generated test cases were tried. This result demonstrates that rule-based test data generation is feasible, and shows great promise in assisting test engineers, especially when the rule base is developed further.

While the above results were impressive, they are not presented to demonstrate the immediate applicability of this rule base or even this paradigm. The rule base needs considerable development and may eventually evolve into a system of hundreds of rules. Similarly, the parser/scanner and test case execution interfaces with the test data generator require considerable development before the paradigm can be fully implemented. However, these can now proceed recognizing the potential that exists as demonstrated by the experiments documented above.

REFERENCES

1. B. Beizer, *Software Testing Techniques*, New York: Van Nostrand Reinhold Company, 1983.

2. D. B. Brown, "Test Case Generator for TIR Programs," Contract Number DAAH01-84-D-A030, *Final Report*, September 30, 1986.

3. 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.

4. J. W. Duran and J. J. Wiorkowski, "Quantifying Software Validity by Sampling," *IEEE Trans. Reliability*, Vol. R-29, No. 2, June 1980.

5. J. W. Duran and S. Ntafos, "A Report on Random Testing," in *Proc. of 5th International Conference on Software Engineering*, Mar 9-12, 1981.

6. R. E. Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, 1985.

7. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Software Engineering*, Vol. SE-1, No. 2, June 1975.

8. W. E. Howden, "Life-Cycle Software Validation," *IEEE Computer*, Vol. 15, No. 2, February 1982.

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

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

11. J. C. Huang, "Program Instrumentation and Software Testing," *IEEE Computer*, Vol. 11, No. 4, April 1978.

12. E. F. Miller, "Software Testing Technology: An Overview," in *Handbook of Software Engineering*, New York: Van Nostrand Reinhold Company, 1984.

13. G. J. Myers, *The Art of Software Testing*, Wiley, New York, 1979.

14. S. C. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Engineering*, Vol. 14, No. 6, June 1988.

15. D. J. Panzl, "Automatic Software Test Drivers," *IEEE Computer*, Vol. 11, No. 4, April 1978.

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

17. S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 4, April 1985.

18. S. M. Ross, "Statistical Estimation of Software Reliability," *IEEE Trans. Software Engineering*, Vol. SE-1, No. 5, May 1985.

19. S. M. Ross, "Software Reliability: The Stopping Rule Problem," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 12, Dec. 1985.

20. U. Voges and J. R. Taylor, "Systematic Testing," in *Verification and Validation of Real-Time Software*, Ed. by W. J. Quirk, New York: Springer-Verlag, 1985.

21. O. R. Weyrich, Jr., D. B. Brown, and J. A. Miller, "The Use of Simulation and Prototypes in Software Testing," in *Tools for the Simulation Profession - Proceedings of the 1988 Conferences*, Orlando, Florida, Society for Computer Simulation.

22. O. R. Weyrich, Jr., S. L. Cepeda, and D. B. Brown, "Glass Box Testing Without Explicit Path Predicate Formation," *Proceedings of the 26th Ann. Conf. Southeast Regional ACM*, Apr 20-22, 1988, Mobile, Alabama.

23. E. J. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Trans. Software Engineering*, Vol. SE-12, No. 12, Dec. 1986.

APPENDIX A

DEFINITIONS OF TERMS

Definitions for the following terms have been adapted from several sources [4, 7, 13, 14, 17, 18, 19].

branch coverage - Achieved when test cases are sufficient to ensure that each branch or decision in a program is executed both true and false at least once (usually includes statement coverage).

condition coverage - Achieved when test cases are sufficient to ensure that each condition in a decision takes on the value of true and false at least once during execution.

debugging - The act of searching for and removing a fault from a program.

decision coverage - See branch coverage.

decision-condition coverage - Achieved when test cases are sufficient to ensure that each condition in a decision and the decision itself take on the values of true and false at least once during execution.

error - An incorrect output resulting from a fault.

fault - An incorrect program component, while an error is an incorrect output resulting from a fault.

instrumentation - The insertion of code at a strategic point in a program for the purpose of reporting (1) when and if execution reaches that point and (2) the values of key variables and/or expressions.

linear path predicate - A path predicate which contains only linear terms.

multiple condition coverage - Achieved when test cases are sufficient to ensure that all possible combinations of conditions outcomes in each decision, and all points of entry, are invoked at least once

oracle - An external source of information capable of determining whether or not a program has executed correctly for a given test case.

path coverage - Achieved when test cases are sufficient to ensure that each path in a program is traversed at least once during execution. This is ususally considered an intractable problem since the number of paths in a program may grow exponentially as a result of loops.

path predicate -The conjuction of all conditions along a given path such that if the predicate is satisfied (i.e., a set of values for the variables is found that yields a value of true for the predicate) then that set of values causes the path to be traversed during execution.

regression testing - Previously executed test cases executed again subsequent to program modification to ensure that new faults were not introduced during the modification process.

software reliability - Usually characterized as mean time between failures in a particular environment over a specified period of time.

statement coverage - Achieved when test cases are sufficient to ensure that each statement in a program is exercised at least once during execution.

test adequacy criteria - A quantifiable measure of the degree to which a program has been tested, e.g., (in increasing order) statement coverage, branch or decision coverage, condition coverage, decision-condition coverage, multiple condition coverage, path coverage.

test case - A set of input values, expected output values, and observed results for one execution of a program.

test data generation - The generation of input values for test cases.

test suite - A set of test cases, usually designed to meet one or more specific test adequacy criteria such as decision coverage.

FOOTNOTES

W. H. Deason is with Intergraph, One Madison Industrial Park, Huntsville, Alabama 35807.

D. B. Brown, K. H. Chang and James H. Cross II are with the Department of Computer Science and Engineering, 107 Dunstan Hall, Auburn University, Alabama 36849.

This work was supported, in part, by a contract with NASA, Marshall Space Flight Center, Huntsville, Alabama.

INDEX TERMS

Software testing, test data generation, rule-based systems, Ada testing, unit level testing, test coverage,

FIGURE CAPTIONS

Figure 1. Rule Based Test Case Generator Paradigm

APPENDIX C

QUEST/ADA: AN AUTOMATED TOOL FOR ADA SOFTWARE TESTING

The paper given in this appendix was produced in part by support provided by this project contract. This paper was published in the Proceedings of the 27th Annual Southeastern Regional Conference of the ACM.

ORIGINAL PAGE IS
OF POOR QUALITY

QUEST/Ada: An Automated Tool for Ada Software Testing

William R. Kelcher
David B. Brown, Ph.D.,P.E.

Auburn University

Abstract

The increasing use of mission-critical embedded software systems has made apparent the need for an effective and efficient approach to software testing. At present a variety of approaches to software testing exist, but existing test methodologies require a great deal of effort to produce any results. Thus, these methodologies are impractical for use on large software systems where the complexity of the programs would require a tremendous expenditure of man-hours. In addition to the cost problem, the conceptual complexity of a large software system would make such manual testing methods prone to human error.

This paper will result on the preliminary design of the Query Utility Environment for Software Testing of Ada (QUEST/Ada), an automated tool for testing of Ada software. Quest/Ada uses an expert system in conjunction with a feedback loop from automatic test execution to produce test data sufficient to provide complete or near-complete decision/ condition coverage for a user-selected Ada software module. The paper will detail the preliminary design of the three major components of QUEST/Ada: The parser/scanner used to instrument Ada code, the rule-based system for test generation, and the user interface.

Introduction to QUEST

QUEST/Ada is an automated software testing tool designed to automatically generate test data for complete or near-complete coverage of Ada programs. The test data generation will be performed by a feedback loop consisting of an instrumented Ada module, a test data coverage analysis module, and an expert system which generates new test cases based on the test coverage data provided by the analysis module. Supporting this feedback loop is the parser/scanner which generates the instrumented Ada code from the raw source code, and a report generator which provides user reports on the effectiveness of the test data. Figure 1 provides an overview of the QUEST system.

The preliminary design of QUEST is being performed by generating a functional prototype of the entire testing tool. The majority of the prototype is being written in C, because of the ease of coding system interfaces and the availability of excellent parser/scanner tools which produce C code. In addition to C, the CLIPS expert system language is being used to develop the test data generator. CLIPS was selected because of its availability, ease of use, speed, and because the source code is available and is itself written in C. The QUEST prototype is being generated on a VAX 11/780 computer.

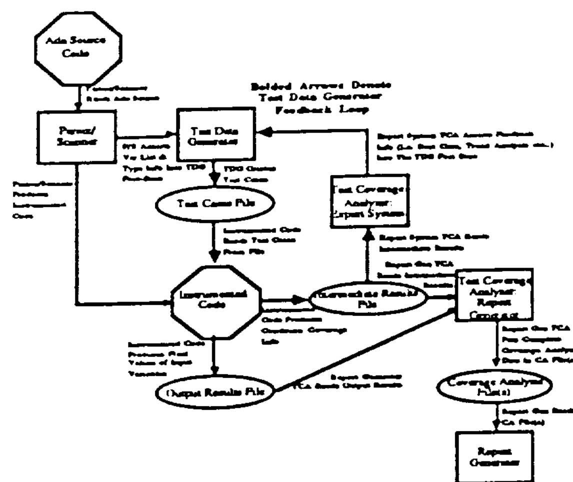


Figure 1: Quest Overview

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©1989 ACM 0-89791-303-5/89/0400-0016 $1.50

The Parser/Scanner

The parser/scanner module is the first functional step in testing a module with QUEST. It is responsible for instrumenting the Ada source code, building a symbol table for the module, and surrounding the instrumented source with a "driver" program which will receive test data from the test data generator and provide execution results to the user interface's report generator. The parser is being developed using the BISON parser generator, which produces a parser written in C.

The primary task of the test data generator is to instrument the source code to provide the data that is required for determining test coverage and providing feedback data for the test data generator. In the prototype, this instrumentation consists of replacing each boolean condition in the module with a function call to the function "relop". This function mimics the boolean condition that it is replacing by examining the operands and operators of the condition and providing the boolean value of the condition as the value returned by the function. More importantly, the "relop" function also writes intermediate test coverage information to a file that is examined by the test coverage analyzer. The intermediate test coverage information provided by the function includes the decision and condition number that the function is replacing, the number of the test case that the instrumented program is currently executing, the boolean value of the condition with its current input data, and the values of the left and right hand sides of the condition.

An example instrumentation for a simple module is described in Figure 2. In this case, the module being instrumented is a procedure, although a module may be a program, procedure, function, block, or a combination of any of the above.

In addition to the instrumentation of the source, the parser/scanner must also generate a symbol table containing information about all of the input variables to a module. Input variables are defined as any variables that are passed to the module as parameters, or any variables that are global to the module, in the case that the module is part of a larger software system. For example, in the simple procedure instrumented above, all of the variables are input variables because all of the variables are passed to the module being tested as parameters to the procedure. However, if a variable "X" was defined within the procedure test and used internal to the procedure, it would not be considered an input variable. The information that must be stored about each input variable includes its name, type, bounds (in the case of subrange types), initial value, and scoping information about the depth at which the variable is defined, in the case that it occurs in more than one procedure in the module under test. The information in the symbol table is used by the test data generator to ensure that the data being generated for a variable is of the right type and does not exceed the bounds of a subrange type. The information in the symbol table is also used by the user interface to determine the names of the variables (as all data internal to QUEST are stored in files that are indexed by position, not name), and to provide the formats in which the data is printed by the report generator. In addition, the parser/scanner uses the symbol table information to generate the driver program for the module under test.

In most cases, a QUEST module will be a subprogram of some sort. In order to execute the subprogram and provide it with the test data generated by the expert system, the parser/

```
procedure test(i: in out integer;
               j: in out integer;
               k: in out integer) is
begin
   while i>j loop
      i:=i-1;
      k:=(k+314) mod 25;
      if i>k then
         while i>k loop
            k:=k+1;
            if k>=27 then
               null;
            else
               null;
            end if;
         end loop;
      else
         if i<k-3 then
            if i-10<j then
               null;
            else
               null;
            end if;
         else
            while i>=k-3 loop
               i:=i-1;
            end loop;
         end if;
      end if;
   end loop;
end test;
```

```
procedure test(i: in out integer;
               j: in out integer;
               k: in out integer) is
begin
   while relop(TestNum,1,0,i,GT,j) loop
      i:=i-1;
      k:=(k+314) mod 25;
      if relop(TestNum,2,0,i,GT,k) then
         while relop(TestNum,3,0,i,GT,k) loop
            k:=k+1;
            if relop(TestNum,4,0,k,GE,27) then
               null;
            else
               null;
            end if;
         end loop;
      else
         if relop(TestNum,5,0,i,LT,k-3) then
            if relop(TestNum,6,0,i-10,LT,j) then
               null;
            else
               null;
            end if;
         else
            while relop(TestNum,7,0,i,GE,k-3) loop
               i:=i-1;
            end loop;
         end if;
      end if;
   end loop;
end test;
```

```
with text_io;
use text_io;
procedure driver is

   TestNum:      integer;
   i,j,k:        integer;
   indata,
   outdata:      file_type;

package int_io is new integer_io(integer); use int_io;

begin
    open(indata,infile,"test.data");
    create(intermediate,out_file,"intermediate.results");
    create(outdata,out_file,"output.data");

    while not End_Of_File(indata) loop
        get(indata,TestNum);
        get(indata,i);
        get(indata,j);
        get(indata,k);

        test(i,j,k);

        put(outdata,TestNum);
        put(outdata,i);
        put(outdata,j);
        put(outdata,k);
        new_line(outdata);
    end loop;

    close(indata);
    close(intermediate);
    close(outdata);
end driver;
```


ORIGINAL PAGE IS
OF POOR QUALITY

scanner must generate a driver program that reads the test data from a file and calls the subprogram with the input test data provided as parameters to the subprogram. As indicated, in Figure 3, development of the driver program relies on the information in the symbol table to provide the number, names, and types of input variables to the module under test.

The Test Data Generator

The test data generator is responsible for generating sets of test data, called "test cases," which attempt to provide complete or near-complete decision/condition coverage of the module under test. Decision/condition coverage is achieved when each boolean condition (i.e."x<5"), and each boolean decision (i.e."x<5 or x>25") has been executed both true and false at least once. The test data generator consists of two main parts: the expert system (written in CLIPS) and the test coverage analyzer (written in C).

The expert system develops test data based upon a set of heuristics and two major lists: the decision/condition coverage list and the best test case list. The decision/condition coverage list contains facts of the form "(decision number, condition number, coverage value)." The decision and condition number are references to the instrumented Ada code described above, and the coverage value is either 0, 1, 2, or 3, representing decisions and conditions that have not been covered, been covered only true, been covered only false, and been covered completely, respectively. The expert system operates by focusing on the first decision that has been covered with only one truth value, and finding the best test case that has executed that decision so far. The best test cases are kept in the best test case list, which contains facts of the form "(decision number, condition number, goodness value, input variable value #1, input variable value #2, ...)." The decision and condition numbers correspond to the decision and condition numbers in the decision/condition coverage list. The goodness value is a value for comparison of test cases executing a given decision or condition that is generated by the test coverage analyzer, and the input variable values represent the actual test case. Once the first partially covered decision has been located and its best test case has been found, the expert system generates new test cases by modifying the elements of the best test case slightly. For example, numerical data might be modified by a percentage of the range of values it could take, such as 2%, and by a small fixed amount, such as .01.

This technique is developed from a technique described by Prather and Myers [PRA87]. The reasoning is that if a particular condition, Cn, is reached, the preceeding conditions, C1 through Cn-1, along the path have also been satisfied. In order to drive the target condition to its' other truth value, all of the preceeding conditions must once again be satisfied. In other words, the inputs are close to the intended goal and a slight modification of the input data is all that is required. Note that by driving the other branch of a decision, other paths and decisions are uncovered which are then treated by the next iteration of the test data generator. The expert system/slight modification technique has been demonstrated to provide coverage that is significantly superior to random test data generation [DEA88].

The test coverage analyzer provides information to the expert system by reading the intermediate results from the execution of previous test cases provided by the function calls in the instrumented Ada code. The test coverage analyzer maintains a list of the decisions and conditions covered by an execution of a test case and their coverage value. It also maintains a list of the best test case for each condition based on the "goodness value" of that test case for that condition. The goodness value is generated according to the following formula:

Selection of test cases based on this formula ensures that the test case selected will be the test case whose right hand side most closely matches its left hand side. The reasoning behind this is that minor modifications to the test case will be most likely to drive the condition with another truth value if the left and right hand sides of the case are very close to each other. After all of the intermediate results have been analyzed, the test coverage analyzer asserts decision/ condition coverage facts and best test case facts into the expert system's fact base. Note that the top two or three test cases for each decision and condition are asserted into the fact base. This prevents the expert system from getting stuck in a loop if the best test case for a condition after the execution of a new set of test data turns out to be the same best test case as before the execution (thus causing the expert system to generate the same data as before, ensuring that the best test case remains the same forever...). In the event that this should happen, the best test case is thrown out, and the next best test case is used to generate a new set of test cases.

The User Interface

The modules described above are the two primary functional units in QUEST. The user interface enables these two modules to be used in an integrated user-friendly environment. The user interface consists of two major parts, the menu screens and the report generator. The menu screens allow the user to perform all of the function required to test a program, and the report generator provides the user with the results of the testing process.

In addition to the main menu, the user interface menu system provides 7 major menus: the system definition menu, the module selection menu, the automatic testing menu, the regression testing menu, the variable definition menu, the testing result report menu, and the utilities menu.
The first three menus each perform one step in defining and testing a module in QUEST. The first step is for the user to define a functionally complete set of Ada source code files using the system definition menu. All source code necessary for the execution of the module to be tested (i.e. the source code for any modules called by that module, the package specifications etc...) must be included by the user at this time. The user interface concatenates all of the files selected into one large file, which is passed to the parser/scanner. The parser/scanner develops the symbol table and a list of the testable modules in the source code. The user then selects one of the modules from the list produced by the parser/scanner, and passes the name of that module back to the parser/scanner. The parser/scanner then instruments that module and develops a driver program for it. The module is now ready for testing. The user goes to the automatic testing menu, selects the maximum number of test cases to run, and presses a function key to start the automatic test generation/execution process. The expert system is called to generate initial test data, then the module driver program is called upon to execute the test cases generated. Finally, the test coverage analyzer is called to compute the coverage and best test data, which it then asserts into the fact base of the expert system, thereby causing the entire process to begin again.

The last four menus all provide access to special capabilities of QUEST. The regression tesing screen allows the user to perform regression testing by executing test cases which have already been generated and executed once. This facility provides for quick testing of minor non-conditional statement changes by re-executing the minimal test set, and provides for mutation testing of the test set itself. The variable definition screen allows users to set any or all of the values of input variables to fixed values, thus providing the capability for users to run their own hand-generated test cases automatically. This facility provides the ability to test the module against the requirements specification without any human error entering the test case execution. The test report menu allows the users the ability to view or print the output of the report generator and finally, the utilities menu performs the same function performed by utilites menus throughout the free world, which is to say any function that does not fit under one of the above headings. Figure 4 provides on overview of the user interface and the system.

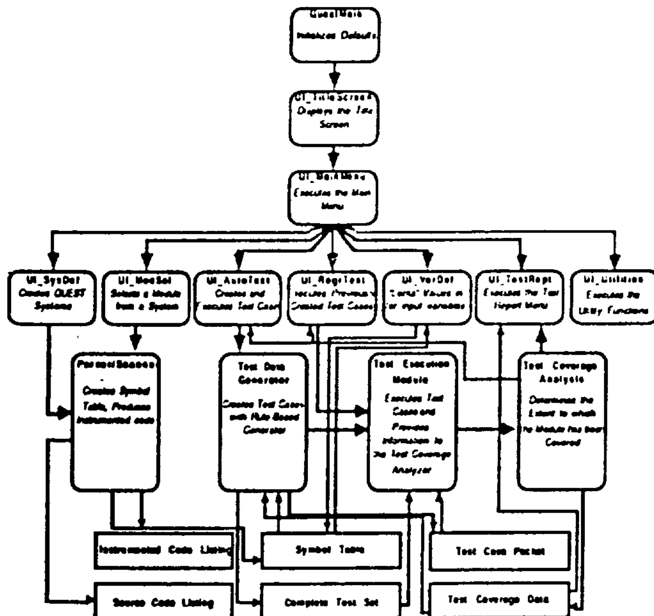


FIGURE 4: Overview of the User Interface

The report generator analyzes test cases, intermediate results and output results of the execution of automatic test cases. The report generator also stores the all of the test cases so that the regression tesing feature may be performed. The report generator has its own test coverage analyzer, because the input requirements to the report generator are fundamentally different than the input requrements of the expert system - the expert system requires mathematical analysis of teporary data, the report generator requires record analysis of permanent data.

Conclusion

In conclusion, it should be noticed that great pains have

ORIGINAL PAGE IS
OF POOR QUALITY

been taken to make the prototype easy to extend, both to other languages and other coverage criteria and testing metrics. No language-dependent information is found in any part of QUEST other than the parser/scanner, other than the assumption of the basic data types supported by every procedural language. All that would be necessary to extend QUEST to another procedural language would be to include another parser/scanner module. This is not a monumental task, as BISON grammars are available in public domain for many popular procedural languages. There is certainly nothing unique to Ada about replacing boolean conditions with function calls, and no language-dependent issues are found anywhere in the user interface - not even in the utilities menu. It is entirely possible for a person with little or no knowledege of Ada to test Ada programs, or programs in any other language, using QUEST. Extending QUEST to other types of coverage criteria and corresponding test metrics would only require placing additional function calls in the appropriate places.

References

[BRO88] Brown, David B. et al, "The Development of a Program Analysis Environment for Ada," Contract Report, NASA Contract: NASA-NCC8-14, Auburn University

[DEA88] Deason, William H., A Rule-Based Software Test Data Generator, Masters Thesis, Auburn University.

[PRA87] Prather, R. E. and Myers, P. Jr., "The Path Prefix Software Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987.

APPENDIX D

EXPERT-SYSTEM ASSISTED TEST DATA GENERATION FOR SOFTWARE BRANCH COVERAGE

The paper given in this appendix was produced in part by support provided by this project contract. This paper has been submitted for publication in Data and Knowledge Engineering.

EXPERT SYSTEM ASSISTED TEST DATA GENERATION FOR SOFTWARE BRANCH COVERAGE

James H. Cross II, Kai-Hsiung Chang,
W. Homer Carlisle and David B. Brown

Department of Computer Science and Engineering
Auburn University, AL 36849

ABSTRACT

With the increased production of complex software systems, verification and validation (V & V) has evolved into a set of activities that span the entire software life cycle. Among these various activities, software testing plays a major role in V & V. Conventional software testing methods generally require considerable manual effort which can generate only a limited number of test cases before the amount of time expended becomes unacceptably large. In this paper, we present a new approach to generating test cases based on artificial intelligence methods. By analyzing the branch coverage of previous test cases, an expert system is able to generate new test cases which provide additional coverage. Heuristic rules are used to modify previous test cases in order to achieve the desired branch coverage. This approach to software testing has the potential for greatly reducing the overall costs associated with branch coverage testing.

Key Words:

Artificial Intelligence, Expert Systems, Knowledge-based Systems, Software Engineering, Software Testing, Test Data generation.

EXPERT SYSTEM ASSISTED TEST DATA GENERATION FOR SOFTWARE BRANCH COVERAGE*

James H. Cross II, Kai-Hsiung Chang,
W. Homer Carlisle and David B. Brown

Department of Computer Science and Engineering
Auburn University, AL 36849

INTRODUCTION

With the increased production of complex software systems, verification and validation (V & V) has evolved into a set of activities that span the entire software life cycle. V & V attempts to ensure that a software system has been designed to meet user requirements and has been implemented according to the design. Testing continues to play a major role in V & V beginning with the development of test plans and test procedures during requirements analysis and design and continuing through the execution of test cases on the implemented system. A variety of approaches to software testing exist [ADR82, GOO75, HOW80, HOW76, HOW82a, WHI80]. However, these methodologies generally require considerable manual effort, i.e., the tester must hand compute paths, predicates, test cases, etc. Manual implementation of these methodologies is not only inefficient in terms of resources expended (man-hours), but it is also subject to inconsistencies brought about by human errors. Manual methods can generate only a limited number of test cases before the amount of time expended becomes unacceptably large. All of these problems may be reduced by the use of automated software test tools. However, automated test data generation itself is not well understood [MIL 84, PAN 78]. In this paper, we present a new approach to generating test cases based on artificial intelligence methods.

Manuscript received ....

* Support for the development of the methodology described in this paper was funded, in part, by George C. Marshall Space Flight Center, NASA/MSFC, AL 35812 (NASA-NCC8-14).

BACKGROUND

The main thrust of software testing research has been aimed at developing more formal methods of software and system testing [BEI83]. By definition, "testing...is the process of executing a program (or a part of a program) with the intention or goal of finding errors" [SHO83]. A test case is a formally produced collection of prepared inputs, predicted outputs, and observed results of one execution of a program [BEI83]. In standard IEEE terminology, a software fault is an incorrect program component; an error is an incorrect output resulting from a fault. In order to detect occurrences of errors indicating faults, some external source of information about the program under test must be present.

Program testing methods can be classified as dynamic and static analysis techniques [RAM75]. Dynamic analysis of a program involves executing the program with test cases and analyzing the output for correctness, while static analysis includes such techniques as program graph analysis and symbolic evaluation [ADR82].

A dynamic test strategy is a method of choosing test data from the functional domain of a program. It is based on criteria that may reflect the functional description of a program, the program's internal structure, or a combination of both [ADR82]. These criteria specify the method of test case generation to be used for a dynamic test strategy. The two dynamic test strategies generally recognized are functional testing and structural testing. It is well-documented in the literature that no one testing approach solves all testing problems; hence, functional and structural testing techniques should be considered complementary methods [HOW80].

Functional Testing. Functional testing involves identifying and then testing all functions of a program (from the lowest to highest levels) with varying combinations of input values to check for correctness of output [BEI84, HOW86]. Correctness of output is determined by comparing the actual output to the expected output computed from the functional

specifications of the program. The specifications are used to define the domain of each variable or its set of possible values. Since the program has input and output variables, selection of test data must be based on the input and output domains in such a way that test cases force (or try to force) outputs which lie in all intervals of each output variable's domain. The internal structure of the program is not analyzed, thus functional testing is often called "black box" testing.

Structural Testing. Structural testing uses the internal control structure of a program to guide in the selection of test data [BEI84], and it is sometimes known as metric-based test data generation. Coverage metrics are concerned with the number of a program's structural units exercised by test data. Test strategies based on coverage metrics examine the number of statements, branches, or paths in the program exercised by test data. This information can be used to evaluate test results as well as generate test data [ADR82]. It can be shown that for many programs (especially programs with loops) the number of possible paths is virtually infinite [BEI84, HOW78, WOO80], thus leading to the problem of determining which paths to choose for testing. Criteria for selecting test paths have been discussed [BEI84, HOW78a, RAM76, SHO83] and include statement, decision, condition, decision-condition, and multiple condition coverage. "Coverage" is said to be achieved if a set of paths executed during program testing meets a given criteria [BEI84]. The problem of finding a minimal set of paths to achieve a particular coverage is discussed in [VIC84] and [NTA79]. Beizer states that the idea behind path testing is to find a good set of paths providing coverage, prove that they are correct and then assume that the remaining untested paths are probably correct [BEI84].

Test Data Generation. Once a set of paths providing coverage has been selected, the next step involves generating test data that will cause each of the selected paths to be executed. Methods for generating test data from paths are discussed in [ADR82, HOW76, HOW75,

HUA75, RAM76] and others, and center around the idea of solving path predicates or at least determining path data constraints to be used for generating test case data.

Generating test inputs for a program may not appear to be a difficult problem since it may be done by a random number generator [DUR81]. However, although random testing alone has been shown to be an inadequate method for exposing errors, when combined with extremal and special value (ESV) testing, it can be an effective method and can provide a direction for the generation of future test cases [VOU88]. On the other hand, algorithms for generating test data to satisfy particular adequacy criteria have generally had poor time and space complexities and produced small amounts of test data. In fact, it is not possible (i.e., there exists no algorithm) to generate test data which causes the execution of any arbitrary program path [MIL84].

DeMillo, Lipton, and Sayward [DEM78] attempted to develop a practical test data generation methodology somewhere between random data generation and full program predicate solution. Noting that programmers produce code that is very close to being correct, they observed a program property which they named the coupling effect. Basically, the coupling effect is the ability of test cases, designed to detect simple errors, to surface more subtle errors as well. Howden, on the other hand, developed a set of functional testing rules [HOW87]. Although both of these research efforts were directed at helping programmers test their code, they are also directly applicable to automatic test data generation. They are not algorithms, but instead are useful rules of thumb. Such rules are typically referred to as heuristics, which embody certain bits of "expert knowledge." Thus, a knowledge-based or expert system approach is very appropriate in attacking the problem of generating test data for software programs. This approach is made possible not only by the maturing body of knowledge about software testing, but also by developments in the field of rule-based systems, a branch of artificial intelligence.

INTELLIGENT TEST DATA GENERATION

The objective of our intelligent test data generation technique is to achieve a maximal branch coverage based on test data generated using heuristic rules with feedback. The analysis of the actual coverage achieved follows the Path Prefix Strategy of Prather and Myers [PRA87]. In the Path Prefix Strategy, a software package is represented in a simplified flow chart. Each condition in the flow chart contains two branches, true and false. The goal of a set of test cases is to maximize the number of covered branches. The coverage of these conditions and branches is recorded in a branch-coverage table. In this table each condition contains two entries, one for the true branch and one for the false branch. When a branch of a condition is covered (or reached), the branch's entry in the table is marked with an "X". In addition to the marking process, the test cases that cover this branch are also recorded. When new test cases are to be generated, we examine the table to select a condition that is not yet fully covered, i.e., only one branch is covered. After a condition is selected, an associated test case of this condition is retrieved. Since the earlier case started from the package entry point and reached the condition under consideration, it is already "close" to covering the branch that is not yet covered. The Path Prefix Strategy uses "inverse" approach to generate a new case. However, as stated in [PRA87], the inverse problem is still not well understood.

Our approach to intelligent test data generation includes the the following tasks: (1) initial case generation, (2) coverage analysis and goodness evaluation, and (3) new case generation. Figure 1 shows the relationships among these major concepts. In this system, a test case is represented as (case number value-1 value-2 value-3 ---- value-n). "Number" indicates the generation sequence of this case. Value-1, value-2, ---, value-n are the values of each input variable of the package. In this initial prototype, we made the following simplifying assumption: a condition contains constants, arithmetic and logic operators, and input variables only. Internal variables, i.e., defined within the tested package, would not appear in a condition. This provides a more explicit relation between the input variables

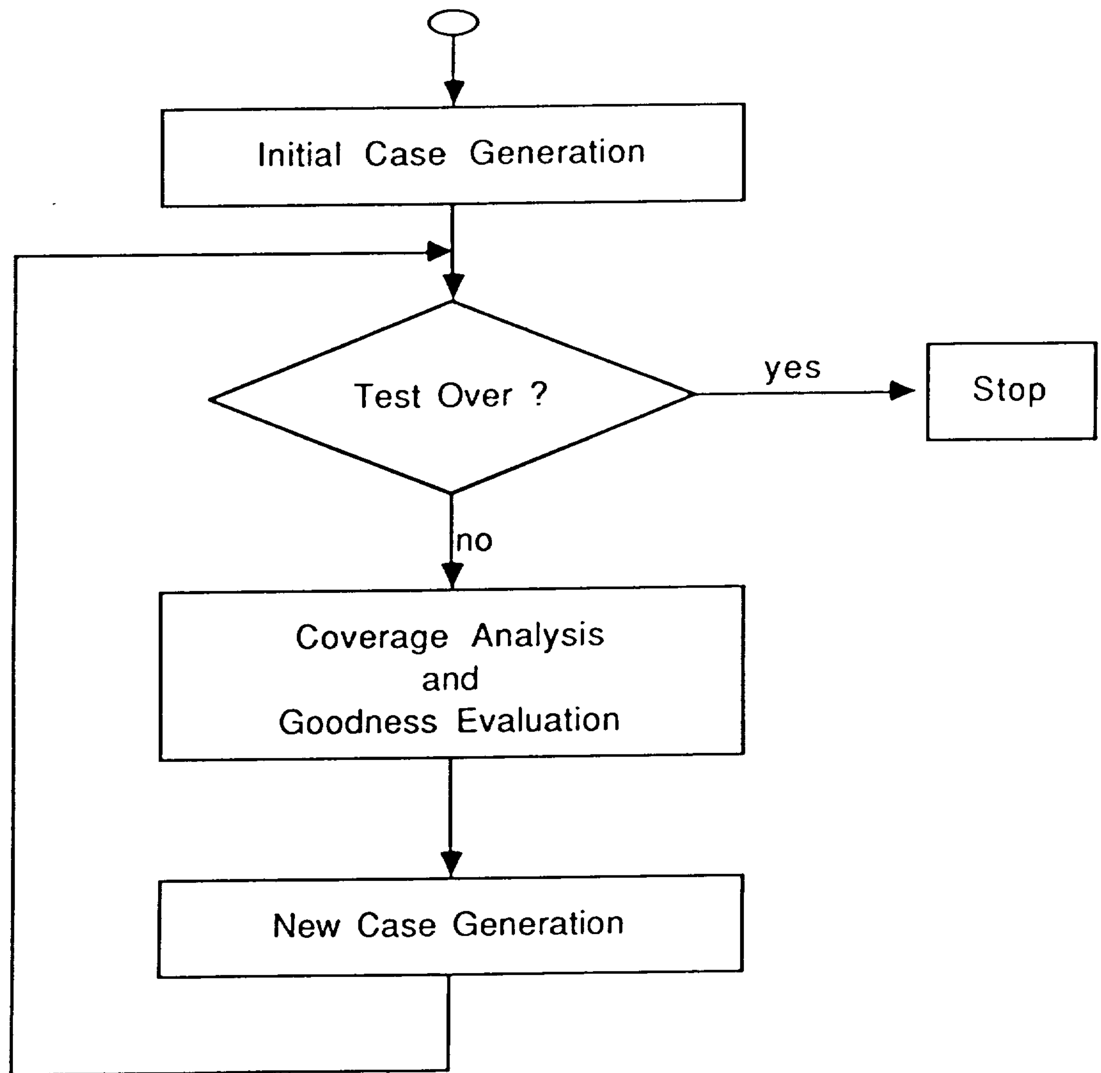


Figure 1 System Concept of the Intelligent Test Data Generator.

and the conditional branches. The system is being implemented using CLIPS [CLI87], an expert system development tool which provides various interfaces to communicate with external functions written in other programming languages, e.g., C, Pascal, Fortran.

INITIAL CASE GENERATION

Initial test cases are needed to start the process. In the event the user provides test cases with the target software package, this initial case generation step can be skipped. However, if test cases are not provided, the initial test cases can be generated based on the syntax information of each input variable, including type, lower limit, and upper limit. Three cases are generated: (1) each variable is assigned to its mid-range, (2) each variable is assigned to its bottom, and (3) each variable is assigned to its top. For each variable, the mid-range, bottom, and top are defined as:

```
mid-range =  (upper-limit - lower-limit) / 2
bottom    =  lower-limit + (upper-limit - lower-limit) * 0.05
top       =  upper-limit - (upper-limit - lower-limit) * 0.05
```

These three cases are numbered cases 1, 2, and 3 respectively. This heuristic rule is written as follows:

Rule-Initial-Cases

```
 (If     no cases exist
  Then   (assert
          (case 1 mid-range-1 mid-range-2 ------ mid-range-n))
         (assert
          (case 2 bottom-1 bottom-2 --------- bottom-n))
         (assert
          (case 3 top-1 top-2 ----------- top-n))
         (assert
          (basket 1 2 3))
         (assert
          (nextcase 4)))
```

If more cases are needed, this rule can be modified to incorporate any desired combinations. In this rule, a basket is asserted to notify that a set of cases has been generated and is ready for coverage analysis. The statement of "nextcase" indicates that the case number of next new case will be 4.

COVERAGE ANALYSIS AND GOODNESS EVALUATION

As indicated above, the objective of the test case generation is to cover as many branches as possible. When a new set of cases is generated, it is analyzed to determine what branches these cases have covered. This process is called coverage analysis and is performed by an instrumented simulator of the target software package. After the coverage analysis, it will be possible to determine what branches have not yet been covered and still need further cases.

The instrumented simulator of a target package functions just like the target package except it contains extra code to record the branch coverage of each condition and to calculate how "close" a test case has been to the condition boundary. For example, an IF statement, IF (3*x+y) >= 21 THEN do-1 ELSE do-2, will be instrumented as IF analyze((3*x+y) >= 21) THEN do-1 ELSE do-2. Here, "analyze" is a function defined in the simulator to perform coverage analysis and other evaluation tasks. The coverage analysis is basically a table filling process which records the covered branches. Assume a test case (x=5, y=2) is analyzed. Since the value of the evaluated left-hand-side (LHS), i.e., 17, is smaller than the right-hand-side (RHS), i.e., 21, the FALSE or the ELSE branch will be executed. This means the ELSE branch is covered.

Besides the coverage analysis, the simulator also calculates how close the LHS is from the RHS, based on the given test case. The "closeness" is defined as

```
ABS(LHS - RHS) / MAX (ABS(LHS), ABS(RHS))
```

This measure tells the closeness of the test case to the condition boundary [DEA88]. When this measurement is small, it is generally true that this test case can be modified slightly to cover the other branch of the condition. In the previous example, with test case (x=5, y=2), LHS is 17 and RHS is 21. The closeness value is therefore (21 - 17) / 21 = 19 %. Assume there is a second test case (x=6, y=2), its closeness value is (21 - 20) / 21 = 4.8 %. The second case will be considered better than the first case. This is because a smaller change on the second case may be enough to lead to the condition boundary or even beyond the boundary and cover the other branch. The importance of the slight modification to the test case is based on the fact that the original case starts from the entry point and reaches the condition under consideration. Between these two points, this case is also used by other conditions. In order to reach the condition under consideration again and cover the other branch, the modified new case must pass through the same set of conditions and yield the same branching results. For this reason, the closeness of a test case (under a particular condition) can also be considered as its "goodness". In the current version of system, the analyzer is written in C language. It is called as an external function from CLIPS.

In this system, each condition contains two pieces of information. The first is the coverage of its branches. The second is the "best" test case that has reached this condition so far. If both branches of a condition have already been covered, this condition will not be considered for further test case generation. However, if only one branch is covered, more test cases need to be generated, based on the best case, to cover the uncovered branch. This heuristic is expressed in the following CLIPS rules. (These rules have been modified slightly for easy understanding.)

```
Rule-No-More-Case

(If    (check ?cond)                  ;check condition ?cond
       (cond ?cond true true $?x)     ;both branches have been
                                      ;covered. Best case, $?x, is
                                      ;ignored.
 then (assert (check (+ ?cond 1))));check next condition.
```

```
Rule-More-Cases

(If    (check ?cond)                ;check condition ?cond
       (not                         ;not both conditions are
        (cond ?cond true true ?x))  ;covered. There is also a
                                    ;best test case, ?x.
 then (assert
        (generate-more ?cond ?x)))  ;pass the condition and the
                                    ;best case to the test case
                                    ;generator.
```

NEW CASE GENERATION

The objective of this portion of case generation is to cover the branches that are not covered previously. As described above, a request of this kind provides the condition to be considered and its best test case. The generation of more test cases should follow the general philosophy of modifying the best case as little as possible. With this guideline in mind, the test data generator (TDG) will generate several sets of new cases with various constraints.

Consider the best test case, case-i, of condition cond-i. Case-i is a list of numerical values (V_1, V_2, ---, V_j, ---, V_n). The j-th value in the list corresponds to the j-th input variable. Since it may happen that only part of the input variables would appear in a condition, the TDG needs to know what variables appear in the condition. The values of these variables have the direct impact of the branching decision and should be the candidates to be modified. Let us assume VL = (V_a, V_b, ---) contains these variables. The following heuristics have been used to generate new test cases.

1. Keep all variable values unchanged except one variable which is in VL. This would simplify the condition under consideration. For example, in the previous best case, (x=6, y=2), if we keep x unchanged, the condition can be simplified in the following steps:

```
a.        3 * x + y  >=  21       ;x=6
```

b. $$18 + y >= 21$$

c. $$y >= 3$$

The last expression tells the condition boundary. Three more new cases can be generated, i.e., (x=6, y=3+e), (x=6, y=3), (x=6, y=3-e). Here, e is defined as a small positive number, e.g., e = (upper-limit of y - lower-limit of y) / 100. The same method is then applied to every variable in VL. If there are m variables in VL, then 3 * m cases will be generated. The simplification process is performed in C- subroutines. This is because CLIPS does not provide symbolic evaluation functions.

2. Keep all variable values unchanged except two variables which are in VL. One of the variables is instantiated to its mid-range value. With this instantiation, only one variable is left in the condition. The simplification process previously mentioned can then be applied. Assume we are considering a condition,

$$4 * x * y + 3 * x =< 9 * y$$

The best test case we have so far for this case is (x=2, y=15, z=10.6). Since only x and y appear in this condition, only their values need to be considered. Also assume the ranges for x and y are [-10.5, 20.5] and [-5.5, 30.5] respectively. Three new test cases can be generated in the following steps.

a. Assign y = mid-range-of-y = [30.5-(-5.5)]/2 = 18

b. Replace every appearance of y in the condition with 18.

$$3 * x * 18 + 3 * x =< 9 * 18$$

$$57 * x =< 162$$

$$x =< 2.842$$

c. Generate 3 cases (x=2.842+e, y=18, z=10.5), (x=2.842, y=18,

z=10.5), (x=2.842-e, y=18, z=10.5).

The system then instantiate x to its mid-range and repeats the process.

These two heuristics will generate many new cases. Additional branch coverage normally can be achieved. However, it may happen that new cases never reach the target condition due to the modification of previous cases. If this happens, the best test case of the target condition would not have been changed. Since the generation of new test cases is based on the best case, the same set of test cases will be generated over and over again and they will never out perform the original best case. The TDG has a mechanism to prevent this fruitless loop. Associated with each condition, there is a "bag". This bag records all best cases that the system has used to generated cases for the condition. As more cases are generated and analyzed, the best case of each condition will evolve. This is the reason that there may be more than one best case in a bag. When the best test case of a condition is found that it has been used before, a fruitless loop may exist. This means previous heuristics did not yield good cases. If this happens, the third and the fourth heuristics will be applied.

3. If there is only one variable in VL, then modify all variable values except the one in VL by 10 % of their ranges and then apply the simplification process described in the first heuristic to find the condition boundary. In a condition with only one variable, the only way of covering both branches is to modify this variable's value according to simplification process of the first heuristic. This is required no matter how a test case reaches this condition. If the modification causes the change of the coverage path, i.e., the condition under consideration can not be reached any more, other variable values must be modified to compensate the change. However, the task of figuring out how a modified variable value would impact the branchings of other conditions and how other variables' value should be modified to compensate the impact is still a future study topic. In this system, all other

variables' values are changed 10 percent of their correspondent ranges to compensate the impact. After the modification to all variables except the one in the condition is made, the simplification step in heuristic 1 is applied to determine its values.

4. If there are more than one variable in VL, heuristic 2 will be applied except all other variables will be modified with 10 percent of their ranges.

TEST DATA GENERATION EXAMPLES

In this section, a simulation of the test data generation process is presented to illustrate the functions of the system. Assume the flow chart of a target software package is given in Figure 2. There are three input variables to the package, i.e., x, y, and z. They are all real numbers and have ranges, [30, 200], [-220, 20], and [-100, 312] respectively. The expressions of the conditions are:

```
Cond-1:        z + 20      <     3 * x
Cond-2:     3 * x * y      =<    4 * y
Cond-3:             z      >     y + 100
```

When this information is presented to the TDG, three initial test cases will be generated based on the first heuristic. They are the mid-range case, (case 1 115 -100 106), the bottom-range case, (case 2 38.5 -208 -79.4), and the top-range case, (case 3 191.5 8 291.4). These cases are then fed to the coverage analyzer. The conditions and branches that each case reaches are:

```
Case-1:    (cond-1 True), (cond-2 True)
Case-2:    (cond-1 True), (cond-2 True)
Case-3:    (cond-1 True), (cond-2 False)
```

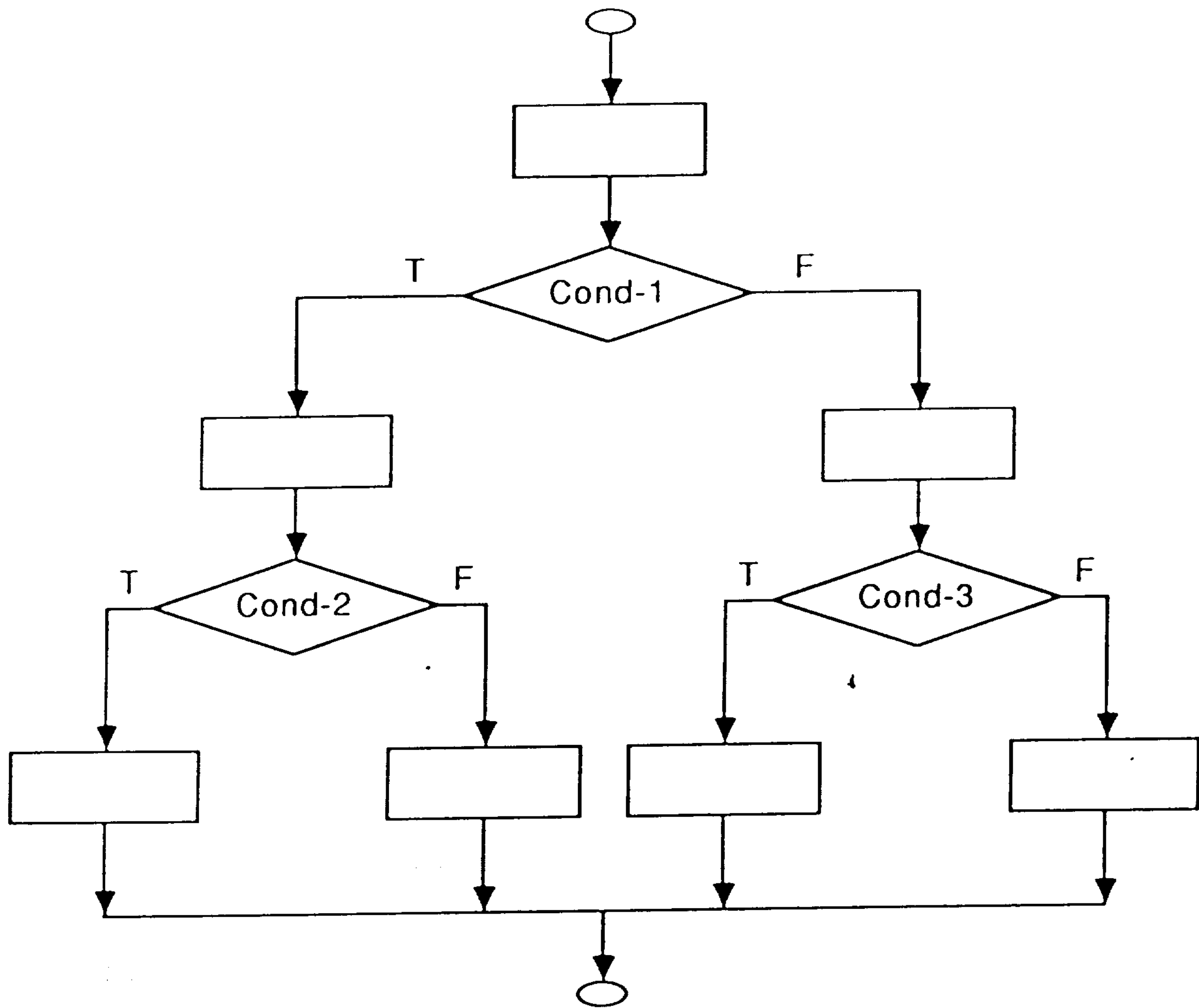


Figure 2 Flowchart of a Target Software Package.

Branch

Condition	T	F
1	X	
2	X	X
3		

Figure 3 The Branch-coverage Table of the Target Software Package after the Initial Cases.

All cases went through the truth branch of cond-1, two cases went through the truth branch of cond-2, and one case went through the false branch of cond-2. The coverage table at this point is shown in Figure 3. Based on the goodness of a case associated with a condition defined earlier, the best test case for cond-1 is case-3 and the best test case for cond-2 is case-2. This information is sent back to the TDG. The TDG will begin by checking cond-1. Since only one branch of cond-1 is covered, more cases should be generated for cond-1. The first and the second heuristics are applied and case-3 is used as the best case. This will generate six new cases. The case generation sequence is:

```
(case 4 191.5 8 554.5)          ;keep x unchanged, modify z only,
                                ;z is out of range, illegal case.

(case 4 103.8 8 291.4)          ;keep z unchanged, modify x only

(case 5 102.1 8 291.4)          ;keep z unchanged, modify x only

(case 6 105.5 8 291.4)          ;keep z unchanged, modify x only

(case 7 115 8 425)              ;keep x at mid-range, modify z only
                                ;z is out of range, illegal case

(case 7 42 8 106)               ;keep z at mid-range, modify x only

(case 8 40.3 8 106)             ;keep z at mid-range, modify x only

(case 9 43.7 8 106)             ;keep z at mid-range, modify x only
```

The process then checks cond-2. Since both branches of cond-2 are covered, no more cases are needed. When cond-3 is checked, the TDG finds that it has not been covered previously. No cases will be generated for cond-3 at this point. The process then passes the newly generated six cases to the coverage analyzer. The analysis result will be:

```
case-4:   (cond-1 False), (cond-3 True)

case-5:   (cond-1 False), (cond-3 True)

case-6:   (cond-1 True),  (cond-2 False)

case-7:   (cond-1 False), (cond-3 False)

case-8:   (cond-1 False), (cond-3 False)
```

```
case-9:    (cond-1 True),    (cond-2 False)
```

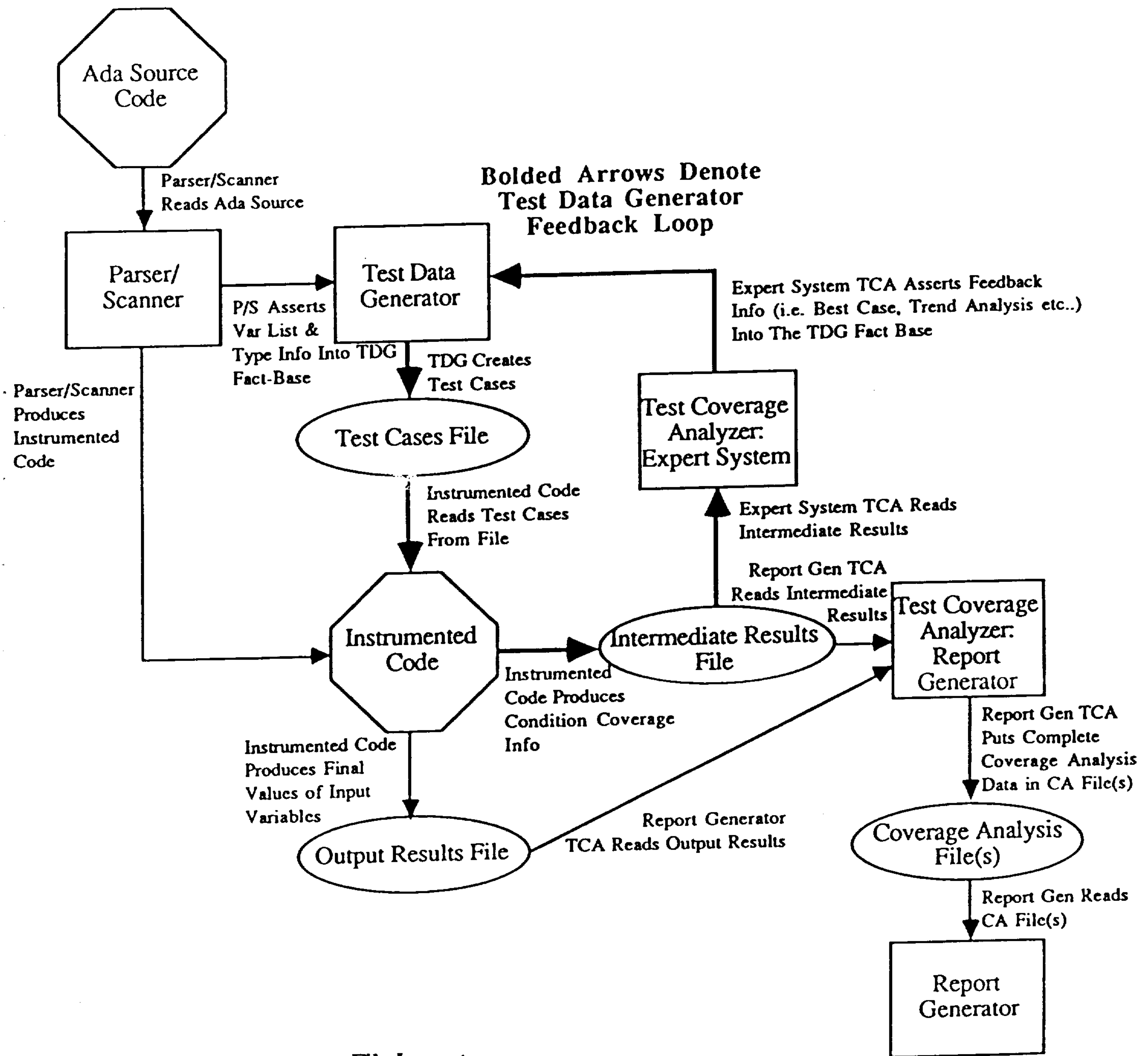
At this point, all conditions are fully covered. The test data generator will stop. Totally, 9 cases are generated.

THE PROTOTYPE

A prototype system, Query Utility Environment For Software Testing (QUEST), has been designed to test Ada programs. Figure 4 provides an overview of the relationships among the major components of QUEST/Ada.

Parser/Scanner. Under control of a user interface module (discussed later) the source code is first processed by the parser/scanner unit. A first pass produces symbol table information for the user interface so that a unit can be selected for testing. Once a unit is selected, the name of that unit is passed back to the parser/scanner. The parser scanner instruments the selected module by replacing conditional expressions with function calls. The function called by the instrumented code evaluates the relational operation and returns the value of the condition. This function call also writes information to files to be examined for coverage by the test coverage analyzer. Information such as the relational operation, the condition number, the decision test case number, the value of the condition and values of the operands to the relational operator are written to files and to the knowledge base of the expert system.

The parser scanner also builds a symbol table to provide information to other units of the system. This symbol table is used to determine those variables of the unit under test that are parameters to the unit or are global variables to the unit. Information concerning the type, ranges, initial values and scoping within the unit is gathered during the parse of the input. The primary user of this information is the expert system test data generator, but the symbol table information is also used for I/O by the user interface and the report generator.



File Layouts:

Test Cases File:	test#, input_value1, input_value2 etc...
Intermediate Results File:	test#, decision#, condition#, T/F, LHS, RHS
Output Results File:	test#, output_value1, output_value2 etc...

Figure 4. Major Components of QUEST/Ada

The parser/scanner itself uses symbol table information to construct a surrounding driver program for the unit under test. Thus all information and statement "hooks" required by other components of the system are inserted into a copy of the original unit. This unit is surrounded by a driver program also written in the unit language enabling the tool to be ported to other environments supporting C, Clips, and the unit's language. This approach also assists the users understanding of the functioning of the system and offers an opportunity to utilize other testing or debugging tools. Since the parser/scanner unit has been implemented using the BISON parser generator and an LALR description of ADA, the tool can be adapted easily for other languages.

Test Data Generator and Test Coverage Analyzer. The primary components of the expert system are the test data generator and the test coverage analyzer. Using the compiled output of the parser/scanner, the test coverage analyzer executes the program for a test case and analyses the results. Based on this analysis, the test data generator creates a new test case. The function and implementation issues of these units were described above.

User Interface. The parser/scanner, the test data generator and the test coverage analyzer components are the primary functional units of the prototype. The user interface is designed to control these units as well as present the results in a comprehensible format. Terminal dependencies are handled by using the VAX-C Curses runtime library routines.

Figure 5 provides an overview of the QUEST user interface and its relationship with other major components and files in the prototype. The user interface main menu provides access to seven subcomponents. The first two components assist the user in defining a functionally complete set of Ada source code files and to determine the unit to be tested. The next component is a testing component that gives the user the opportunity to select options such as the number of test cases to run, and to initiate the automatic test/analyze loop. This selection causes the expert system to generate initial test data and to call the

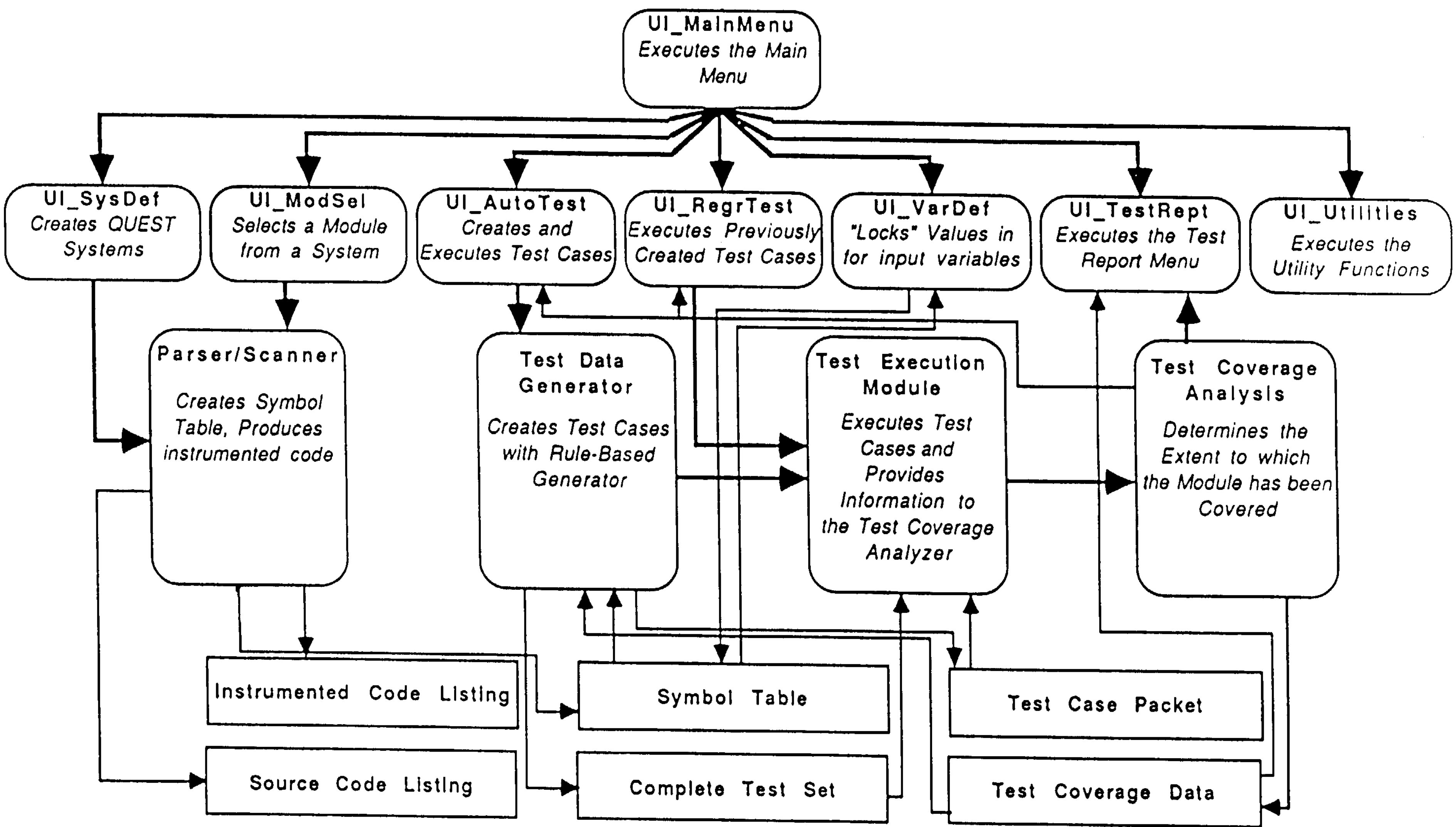


Figure 5. User Interface of QUEST/Ada

driver module to execute the test cases. The results are written to the knowledge base of the system and the process can continue. The final four components of the user interface provide the user with special capabilities of the prototype. These include regression testing, direct manipulation of input values, report generation, and miscellaneous system utilities.

CONCLUSION

The objective of our research has been to explore the potential of expert system based approaches to testing. By combining previous software coverage analysis techniques and the artificial intelligence knowledge-based approach, more efficient test data generation can be achieved. When more test cases are needed to cover a specific branch, heuristics are used to simplify the problem of finding the condition boundaries from which new test cases can be defined. We believe this approach provides a viable alternative with significant advantages over traditional methods for attaining branch coverage.

The overall design of the prototype has been completed and we have implemented the critical components of the system in order to demonstrate the feasibility of this rule-based approach. Although the heuristics presented in this paper have been proven to be useful, additional rules must be formulated to improve the utility and efficiency of the system as well as to handle more complex code. For example, (1) a condition may contain internally defined variables and (2) input variables may be modified. We are currently in the process of completing the prototype implementation to facilitate an indepth comparison of this approach with other more traditional methods and techniques for achieving software branch coverage.

ACKNOWLEDGMENTS

We would like to express our appreciation to each of the project participants for their contributions: William H. Deason, John R. Huggins, William R. A. Keleher, Michael P. Woods, Todd Blevins, and Edward Swan.

REFERENCES

[ADR82] Adrion, W. Richards, et al., "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, Vol. 14, June 1982.

[BEI83] Beizer, B., *Software Testing Techniques*, New York: Van Nostrand Reinhold Company, 1983.

[BEI84] Beizer, B., *Software System Testing and Quality Assurance*, New York: Van Nostrand Reinhold Company, 1984.

[CLI87] *CLIPS Reference Manual*, Version 4.1, Artificial Intelligence Section, Johnson Space Center, NASA, September 1987.

[DEA88] Deason, William H., "Rule-Based Software Test Data Generation," *Thesis*, Auburn University, December 1988.

[DEM78] DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, Vol. 11, No. 4, April 1978.

[DUR81] Duran, J. W. and Ntafos, S., "A Report on Random Testing," in *Proceedings of the 5th International Conference on Software Engineering*, March 9-12, 1981.

[GOO75] Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[HOW75] Howden, W. E., "Methodology for the Generation of Program Test Data," *IEEE Transactions on Software Engineering*, Vol. C-24, May 1975.

[HOW76] Howden, W. E.,"Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, Vol. SE-2, September 1976.

[HOW78] Howden, W. E., "Theoretical and Empirical Studies of Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-4, July 1978.

[HOW80] Howden, W. E., "Functional Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-6, March 1980.

[HOW82] Howden, W. E., "Life-Cycle Software Validation," *IEEE Computer*, Vol. 15, No. 2, February 1982.

[HOW86] Howden, W. E., "A Functional Approach to Program Testing and Analysis," *IEEE Transactions on Software Engineering*, Vol. SE-12, October 1986.

[HOW87] Howden, W. E., *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

[HUA75] Huang, J. C., "An Approach to Program Testing," *ACM Computing Surveys*, Vol. 7, September 1975.

[MIL84] Miller, E. F., "Software Testing Technology: An Over view," in *Handbook of Software Engineering*, New York: Van Nostrand Reinhold Company, 1984.

[NTA79] Ntafos, S. C. and Hakimi, S. L., "On Path Coverage Problems in Digraphs and Applications to Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-5, September 1979.

[PAN78] Panzl, D. J.,"Automatic Software Test Drivers," *IEEE Computer*, Vol. 11, No. 4, April 1978.

[PRA87] Prather, R.E. and Myers, P., Jr., "The Path Prefix Software Testing Strategy," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 7, July 1987, p. 761-765.

[RAM75] Ramamoorthy, C. V. and Ho, S. F., "Testing Large Soft ware with Automated Software Evaluation Systems," *IEEE Transactions on Software Engineering*, Vol. SE-1, March 1975.

[RAM76] Ramamoorthy, C. V. et al., "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering*, Vol. SE-2, December 1976.

[SHO83] Shooman, M. L., *Software Engineering*, New York: McGraw- Hill Book Company, 1983.

[VIC84] Vick, C. R., and Ramamoorthy, C. V., *Handbook of Soft ware Engineering*, New York: Van Nostrand Reinhold Company Inc., 1984.

[VOU88] Vouk, Mladen A., McAllister, David F., and Tai, K. C., "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software," in *Workshop on Software Testing Proceedings*, IEEE Computer Press, 1986.

[WHI80] White, Lee J. and Cohen, E. I., "A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-6, May 1980.

[W0080] Woodward, M. R., et al., "Experience with Path Analysis and Testing of Programs," *IEEE Transactions on Software Engineering*, Vol. SE-6, May 1980.

APPENDIX E

DESIGN OF THE USER INTERFACE

A concerted effort was made to separate the user interface design documentation from the other parts of the design. This was done to eliminate the complexity that would result, making the diagrams virtually unreadable. For this reason the user interface is omitted from the IORL system description given in Appendix A.

This is not to minimize the importance of the user interface design. In fact, as the user interface evolved it contributed heavily to the system structural design. Further, the user interface is important from the standpoint that QUEST/Ada will be worthless unless it can be operated easily by Ada code test personnel.

The user interface presented in this section is expected to continue to evolve throughout Phase 2 of the project. Until such modifications are made, it can be used as a user manual for the QUEST/Ada system. Figure E.1 gives an overview of the user interface as it interacts with the four components of the system (compare with the IORL SBD, Document: QUEST).

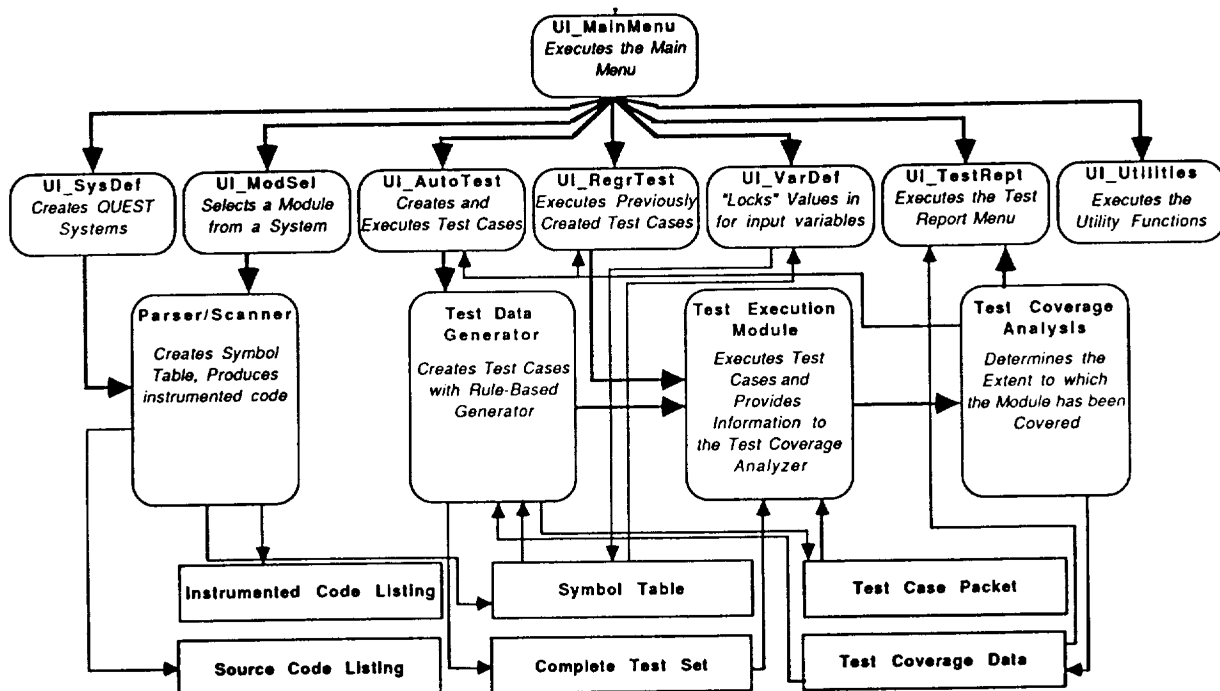


Figure E.1 Overview of User Interface

The QUEST Prototype is invoked by typing "QUEST" at the DCL command prompt. This executes a command file which establishes the environment for QUEST, and then starts the system. The first screen to appear is the title screen; press <ENTER> to go on to the main menu, which is given in Figure E.2. This is the overall controlling menu for the system. It will appear when QUEST is invoked from the operating system. Each entry of this menu corresponds to a function in Figure E.1.

Note first that there are seven major options on the main menu which are chosen by typing the corresponding digit 1-7. There are also two other options, Help and Exit, which are available from any QUEST screen by using the <HELP> and <PF4> keys. The box at the bottom of the screen displays the current system and ADA module under test.

The menu items are arranged in the order that they are usually invoked, as described below:

1. System Definition. As a first step, the system containing the module to be tested must be identified to QUEST.

2. Module Selection. Once a system is identified, the module to test in that system must be selected.

3. Automatic Testing. This is the actual testing of the module, which involves a number of steps described below.

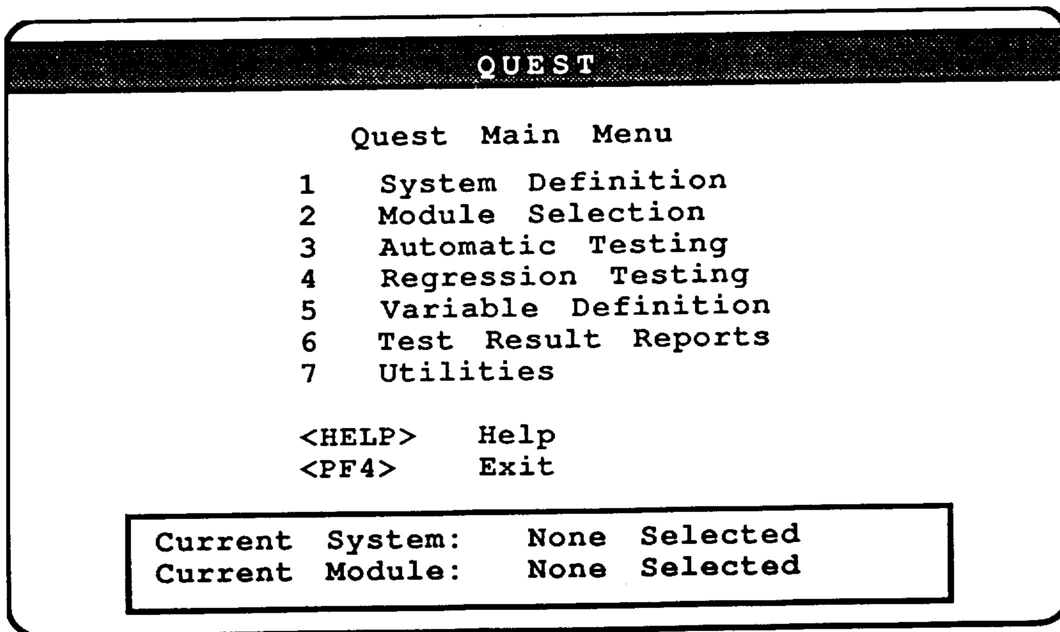


Figure E.2 QUEST Main Menu

4. Regression Testing. This is only performed after module modification, usually as a result of debugging. It has the effect of repeating all previous tests and automatically determining if there were any errors introduced by the modification. This is essential to assure that bugs were not introduced by debugging, especially in areas other than that addressed by the error removal process.

5. Variable Definition. This function allows the user to fix the value of variables in the test. Since any good software design will have predefined tests designed for verification and validation, QUEST allows these to be entered at this point.

6. Test Result Reports. This leads to a series of optional coverage reports which display details of the tests performed at any given point in the testing process.

7. Utilities. This entry contains some housekeeping functions and other options which do not logically fit into the other menu selections.

Each of these menu entries will be described in a separate subsection below which corresponds to the number of the entry in Figure E.2.

E.1 SYSTEM DEFINITION MENU

System Definition is used to identify to QUEST the collection of software modules which is to be subjected to test. This collection of inter-related software henceforth will be called the "system." In this context, a system is a complete functional collection of Ada source code files. Therefore, all modules necessary for executing any of the units to be tested must be included in the system at this time. Figure E.3 illustrates the screen which appears when this option is chosen from the QUEST Main Menu.

The source files which appear in the large window are from the current source directory (which can be altered by Option 7 on the main menu). Source files are selected by moving the cursor through the list using the cursor keys, and then pressing <SELECT> to include a file in the system. The entry of <SELECT> on a previously included file will remove it from the list. The selected files are highlighted on the screen. In the example of Figure E.3, the user moved the cursor to SOURCE4.ADA, entered <SELECT>, and did likewise with SOURCE5.ADA, thus selecting these two files for the system under consideration.

When finished selecting the files, Option 1 on this menu initiates the actual creation of the system. The window at the

bottom of the screen prompts for a name to be assigned to this system, as illustrated in Figure E.4. The notice "Working..." will the appear in the window until the system has been created, as shown in Figure E.5.

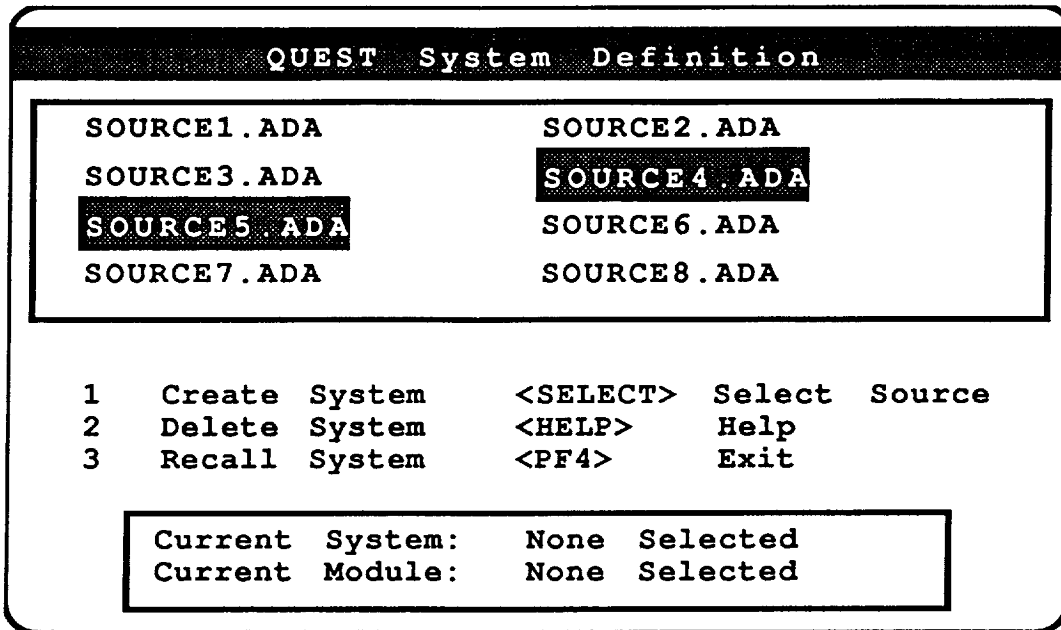


Figure E.3 QUEST System Definition Menu

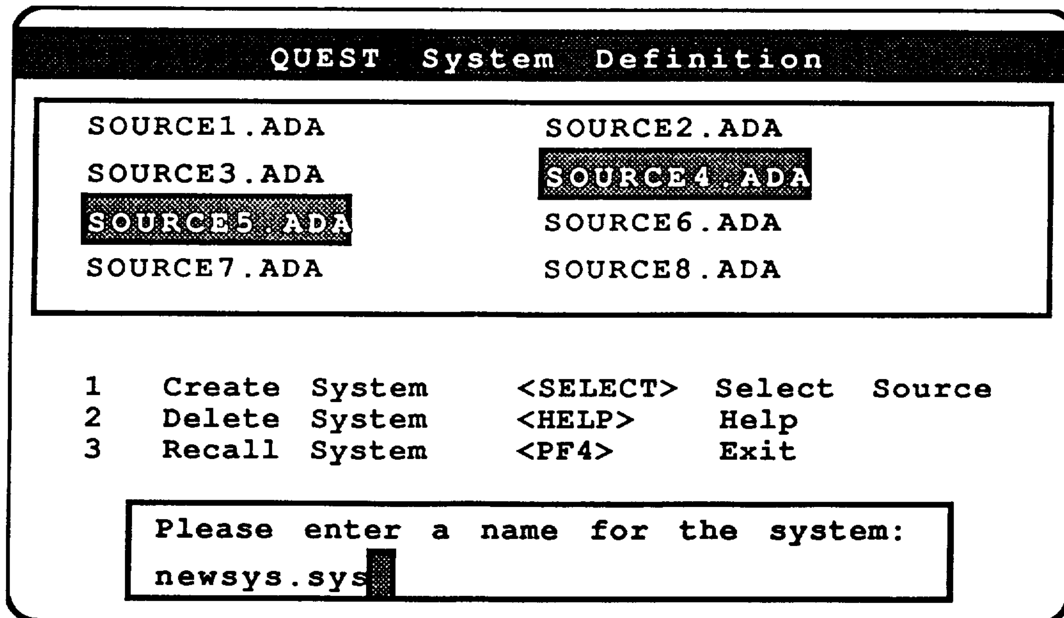


Figure E.4 QUEST System Definition Menu -- Naming

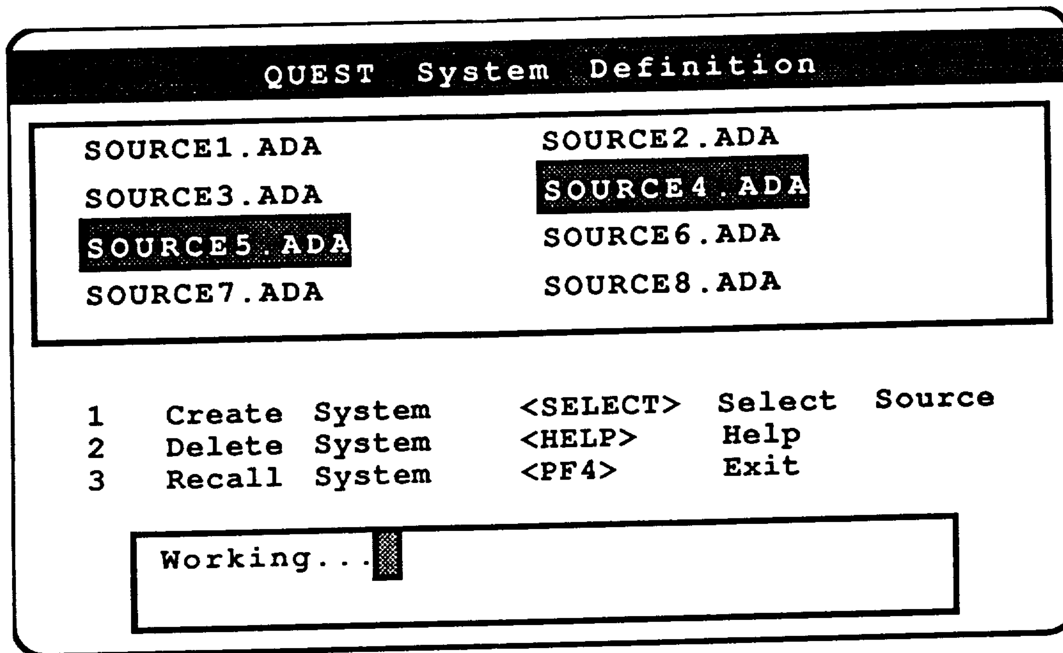


Figure E.5 QUEST System Definition Menu -- Working

As an alternative to directly selecting the files to be included, Option 2 of this menu presents a list of previously defined systems. The choice may be made in a manner similar to that described above (i.e., move the cursor and press <SELECT>). Option 3 presents the same list, but the system chosen is a system to be deleted. To return to the main menu, use PF4.

E.2 MODULE SELECTION MENU

Option 2 of the Main Menu enables a selection of the module to be tested. A list of all modules in the previously-defined system will be displayed in the large window of the Module Selection Menu. Modules for test are selected by moving the cursor to the desired module name and pressing <SELECT>. In Figure E.6, Module 8 is being selected. The selection of a module will automatically bring back the Main Menu.

E.3 AUTOMATIC TESTING MENU

Initially, the Automatic Testing Menu appears as shown in Figure E.7. To initiate a test of the current module, use option 1 -- Begin Testing. QUEST responds by asking for the maximum number of test packets to be generated. Once the desired number is entered, testing begins. The maximum number of packets to be generated is displayed at the top of the screen. Below it, the

Test Data Generator updates the number of packets that it has created, and the last test that it created. Immediately below that is the coverage information. This includes the last test updated, as well as the percentage of coverage achieved so far. The window at the bottom displays various messages about the progress of the testing.

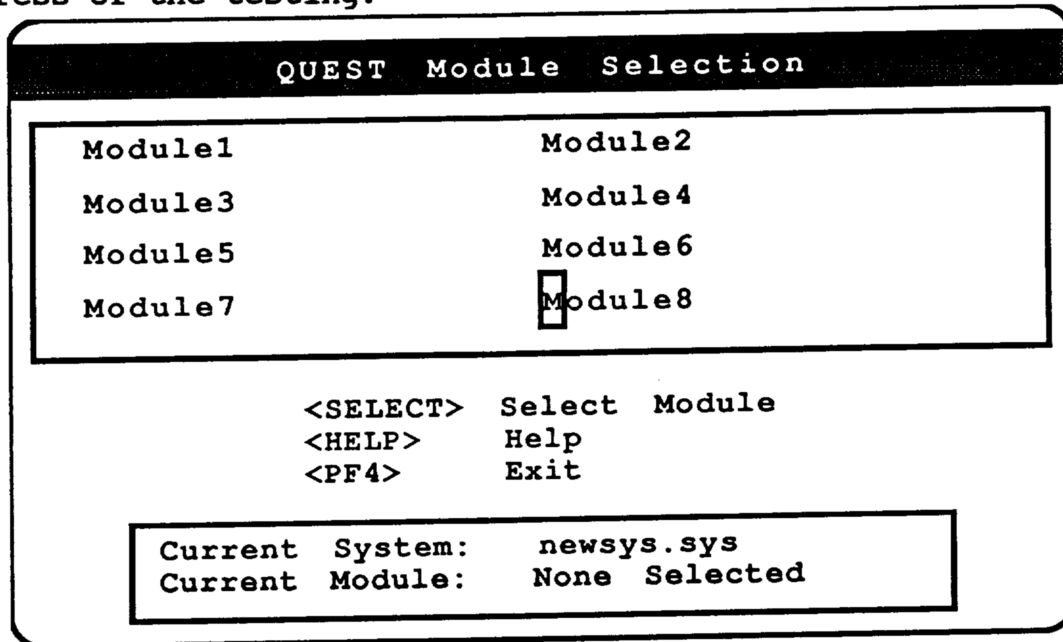


Figure E.6 QUEST Module Selection Menu

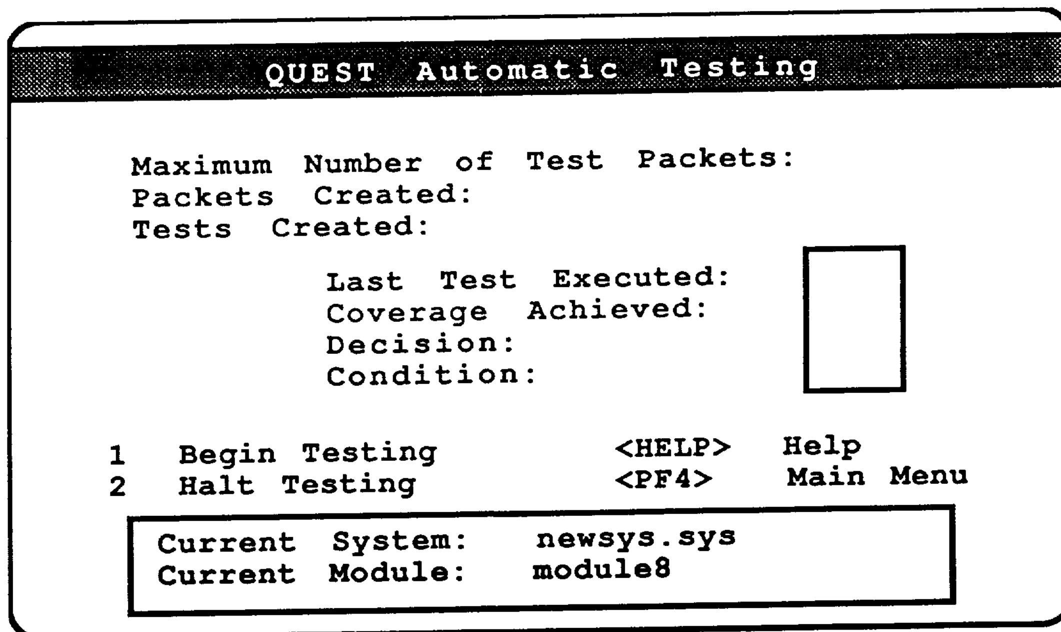


Figure E.7 QUEST Automatic Testing Menu

To stop the testing, enter 2. To return to the main menu, press <PF4>. Help is available throughout the QUEST system as indicated.

E.4 REGRESSION TESTING MENU

Regression Testing allows the user to replay a test of a module. This is essential after any program modification to assure that errors have not been introduced during debugging. This menu is identical to the Automatic Testing Menu given in Figure E.7 above, with the exception that the user is not prompted for the maximum number of test packets. Also, the test generation data (the top three lines) display the final information from the original test.

E.5 VARIABLE DEFINITION MENU

Option 5 of the Main Menu brings up the screen given in Figure E.8. This menu enables values to be defined for any or all of the input variables of the module under test. This process will henceforth be referenced as "locking" the variable, as it prevents the Test Data Generator from creating values for those variables. When the Variable Definition screen is initially displayed, the variables recognized as input variables by QUEST are displayed in the large window.

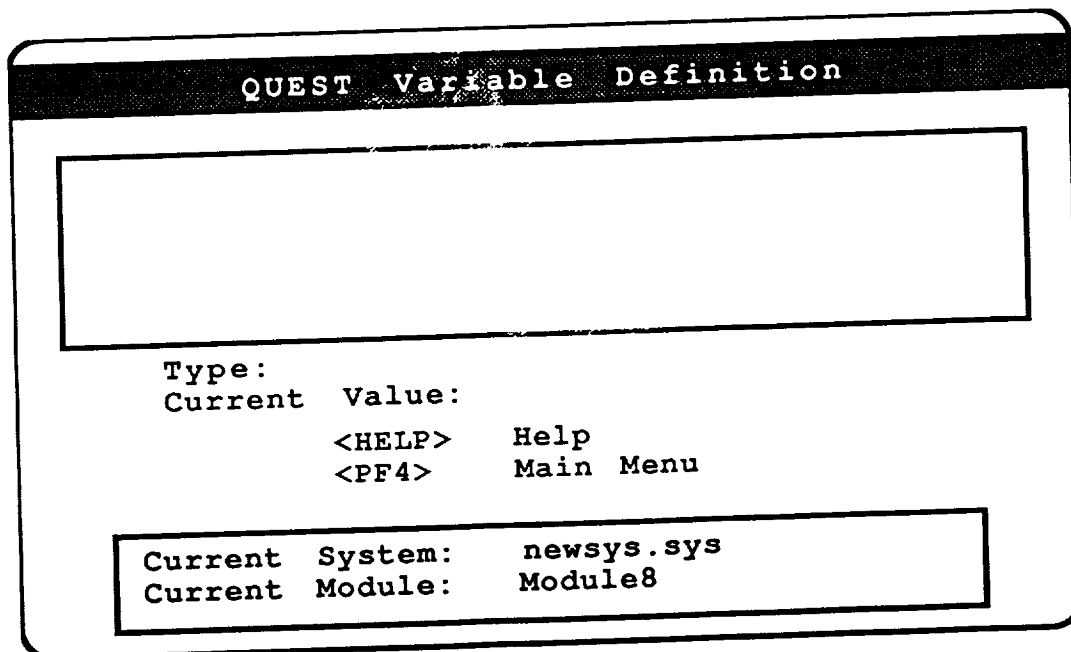


Figure E.8 QUEST Variable Definition Menu

Any variables that are composite types (such as arrays and records) are denoted with a "+" to the left of the variable name. If a composite variable is selected, the name of that variable is placed in the upper text window and the variable's components (i.e. fields in a record, elements in an array, etc...) are placed in the main text window. The user can descend as far as the composite type allows, and can return to the depth immediately above the current depth by selecting the "UP" marker that appears in the top left of the main text window for every composite variable.

Variables that are currently user defined are marked with an "*" to the left of the variable name. A variable for definition may be selected by highlighting it with the arrow keys and pressing return. When a variable is selected, its type, scope, and current user-defined value (if any exists) are displayed on the screen. A new value for that variable can then be entered in the "New Value" field.

E.6 TEST RESULT REPORTS MENU

The Test Result Reports Menu is shown in Figure E.9. The three reports that can be viewed are the Test Coverage Report, the Cumulative Coverage Report, and the Regression Test Report. Selecting any of the reports will bring up the text of the appropriate report. Cursor keys may be used to scroll through the listing, and the <PF4> key will return to the Test Result Reports Menu.

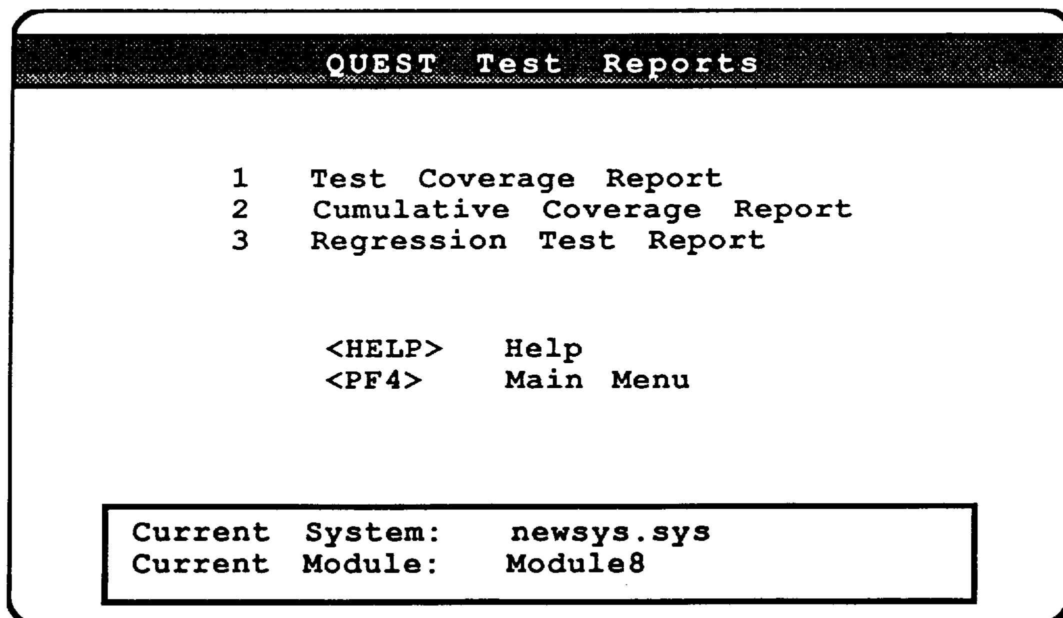


Figure E.9 QUEST Test Result Report Menu

E.7 UTILITIES MENU

This menu includes a complete list of the housekeeping and other odd functions of QUEST, including changing directories, renaming, copying, and deleting files, setting defaults, and printing files. Because this menu is evolving with the design and is expected to change so dramatically, it is not presented in detail at this point.

E.8 SUMMARY OF USER INTERFACE DESIGN

The documentation given above has formed the basis for an early user interface which will facilitate the remainder of the design and development of the other component prototypes. For this reason this portion of the design/development has been allowed to lead the others. Recognize that many modifications of the user interface design are expected. The documentation in this section will continue to be modified and heavily augmented during prototype development to form the final user manual.

APPENDIX F

PROPOSAL FOR PHASE II

This appendix presents the proposal for Phase II which was submitted to NASA in early May of 1989 in order to form the basis for the approval of Phase II funding.

PROPOSAL

THE DEVELOPMENT OF A PROGRAM ANALYSIS ENVIRONMENT FOR ADA (PHASE 2)

Submitted to

National Aeronautical and Space Administration

Marshall Space Flight Center

Submitted by

Department of Computer Science and Engineering
107 Dunstan Hall
Auburn University, Alabama 36849

April 25, 1989

David B. Brown, Ph.D., P.E.
Professor and Acting Head
Computer Science and Engineering
Co-Principal Investigator

James H. Cross II, Ph.D.
Assistant Professor
Computer Science and Engineering
Co-Principal Investigator

M. Dayne Aldridge, Ph.D.
Associate Dean and Director
Engineering Experiment Station

Paul F. Parks, Ph.D.
Vice President for Research

THE DEVELOPMENT OF A PROGRAM ANALYSIS ENVIRONMENT FOR ADA (PHASE 2)

This proposal is for Phase 2 of a research project which was initiated on June 1, 1988. This phase of the project consists of two tasks: (1) the development of testing tools, and (2) the development of graphically-oriented reverse engineering tools. Both of these tasks will focus on Ada as the primary high level language for which prototypes will be developed. The proposal is organized by the tasks given above. Within each, the overall goals are presented, followed by the research approach to be applied. This is followed by an overall project budget.

TASK 1 TESTING TOOLS FOR ADA SOFTWARE

TASK 1: GOALS

The primary goals of software support tools for Ada are to improve software quality and reliability as well as increasing development efficiency. Phase I of the current project has been to design and prototype an environment to facilitate expert system assisted testing of Ada code. A formal grammar specification of Ada and a parser generator were used to build an Ada source code instumenter. Rule-based techniques provided by the expert system tool, CLIPS, were used as a basis for the expert system. The prototype 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.

The current prototype for condition coverage provides a platform that represents an expert system interaction with program testing. This expert system can modify data in the instrumented source code in order to achieve coverage goals. Given this prototype it is now possible to evaluate the rule base in order to develop improved rules for test case generation. As the environment matures it is expected that it will become increasingly user friendly.

The goals of Task 1, Phase 2 are the following: (1) to continue to develop and improve the current user interface to support the other goals of this research effort, (2) to 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. The proposed approach to achieving these goals will now be discussed.

TASK 1: RESEARCH APPROACH

This phase of the research includes the following subtasks.

1. **Testing and evaluation of the present prototype.**

The effectiveness of different test case generation rule sets will be evaluated through empirical studies. In order to accomplish this, a performance evaluation mechanism must first be designed and developed. Performance can then be measured in terms of the number of covered condition branches, the percentage of new test cases that cover anticipated branches, and other possible coverage and efficiency metrics. These measurements will be performed for various test case generation rules and different best test case selection policies.

2. Investigations into extension of new test case generation rules.

The current generation of new test cases is based on the "best previous case." A newly generated set of cases may never out perform the best previous case, or it may not reach the designated condition. Under these situations, the best case of a condition would remain unchanged. This will result in fruitless regeneration of a same set of cases. New rules must be added to redirect the case generation effort. Alternatives include larger modification to non-dominant variables and reinstantiation of variables that distract a test case from its designated path.

A second area that requires attention is the new case generation rule set. Currently the new case generation philosophy is to find condition boundary and to generate cases that are either right on the boundary or that are slightly off the boundary. Attention is solely placed on the condition under consideration. It will be more effective if the path that the new cases are anticipated to drive is also considered. This effort will include determining conditions that are on the path and obtaining values for the variables that are used by these conditions. The new case generation rules should avoid modifying variables that appear in conditions that are on the designated path.

3. Investigation into extensions to other testing strategies with expert system support.

This extension is to include parameters that are returned by external functions in the case generation rules. The current prototype considers only input parameters for test case generation. Parameters returned by other functions (e.g., a number typed by an operator on the terminal) may affect condition branching. Study will be done to investigate how the externally defined parameters can be included in the test case generation.

4. Investigation into extensions to integration testing and concurrency.

Integration testing includes combinations of (1) subprograms, (2) tasks, and (3) packages. In particular, successful testing of Ada code must include the testing of the multiple threads of control associated with Ada tasks. Difficulties in such testing arise from the fact that in addition to data dependencies, real time tasking behavior can affect statement coverage. A testing environment for Ada programs must have the ability to simulate or control not only data driven execution paths, but also paths determined by synchronizations and communications.

5. Establish contacts with NASA subcontractors who are actively developing Ada software.

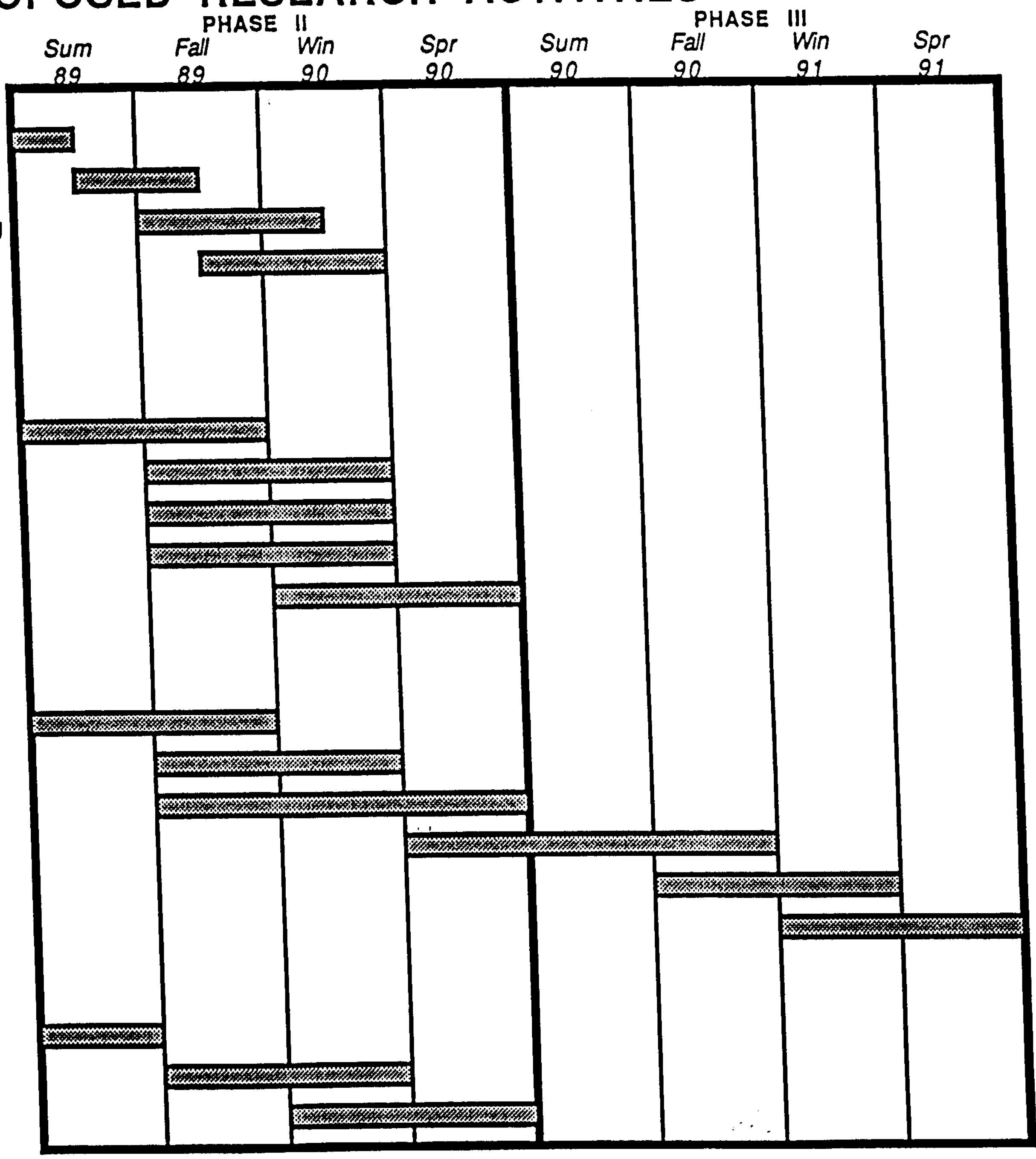
NASA subcontractors have expressed interest in the prototype developed in Phase I. Interacting with these contacts will provide input regarding testing problems

unique to NASA applications, provide a basis for refinements consistent with their needs, and facilitate eventual technology transfer.

TASK 1: PROPOSED RESEARCH SCHEDULE

The following Gantt chart provides the sequence of Task 1 activities to be accomplished during Phase 2 and Phase 3 of this project. Details for the Phase 2 activities were presented above. Phase 3 activities will be developed in greater detail as Phase 2 progresses.

TESTING TOOLS FOR ADA SOFTWARE
PROPOSED RESEARCH ACTIVITIES



TASK 2
REVERSE ENGINEERING TOOLS FOR ADA SOFTWARE

TASK 2: GOALS

Computer professionals have long promoted the idea that graphical representations of software are extremely useful as comprehension aids when used to supplement textual descriptions and specifications of software, especially for large complex systems. The general goal of this research is the study and formulation of *graphical representations of algorithms, structures, and processes for Ada* (GRASP/Ada). The research is presently focused on the extraction and generation of graphical representations from Ada source code to directly support the process of *reverse engineering*.

Our primary motivation for reverse engineering is increased support for software reusability and software maintenance. While applications written in Ada may seem somewhat young to benefit from reverse engineering, NASA and others are quickly amassing libraries of Ada packages. Both reuse and maintenance should be greatly facilitated by automatically generating a set of "formalized diagrams" to supplement the source code and other forms of existing documentation.

The goals of Task 2, Phase 2 are the following: (1) to study, formulate and evaluate graphical representations for Ada software, (2) to develop a prototype reverse engineering tool that includes support for generation of both algorithmic and hierarchical diagrams, and (3) to investigate the generation of additional graphical representations to provide task, package, and data flow views of Ada software. The subtasks outlined in the research approach below are expected to provide a basis for a methodology for graphically-oriented reverse engineering of Ada software.

TASK 2: RESEARCH APPROACH

This phase of the research includes the following subtasks.

1. Formalize a set of graphical representations that directly support Ada software at various levels of abstraction, e.g., system, architectural, and algorithmic levels.

A small, but representative, Ada program will be utilized to formulate and evaluate a set of graphical representations. Specifically, the feasibility of reverse engineering the diagrams from Ada source code will be evaluated. These graphical representations are expected to undergo continual refinement as the automated tools that support them are developed.

2. Design and implement a software tool for generating control structure diagrams (CSDs) and hierarchical diagrams (HDs), e.g. structure charts, from Ada source code.

The present prototype which has focused on CSDs will be extended to include hierarchical diagrams. This subtask will include (1) development of procedures for

identifying and recording module interconnections, (2) development of algorithms for hierarchical diagram layout, and (3) development of methods for displaying/printing hierarchical diagrams on hardware available for this research. The tool will be used on representative Ada software. The generated set of graphical representations will be evaluated for completeness, correctness, and general utility as an approach to reverse engineering.

3. Design and implement the system dictionary component of the GRASP/Ada environment.

This subtask is expected to be done in parallel with the CSD/HD generator subtask and will include (1) development of routines to capture general symbol table information, (2) specification of appropriate report formats, and (3) development of the routines to produce reports from the system dictionary. The system dictionary component is expected to play a major role in supporting the next level of graphical representations of Ada.

4. Investigate additional automatically generated graphical representations of Ada software such as a task view, package/object view, and data flow view.

The task view is expected to be a non-algorithmic view in which task dependencies and interfaces with other Ada components are indicated in a network of communicating processes. This view should provide the user with insight into the concurrent aspects of the Ada software. The package/object view will indicate dependencies among packages as well as data types and operations (functions and procedures) provided by each package. This view will become increasingly important as object-oriented design becomes more widespread. A general data flow view of the software is expected to be the most difficult to generate.

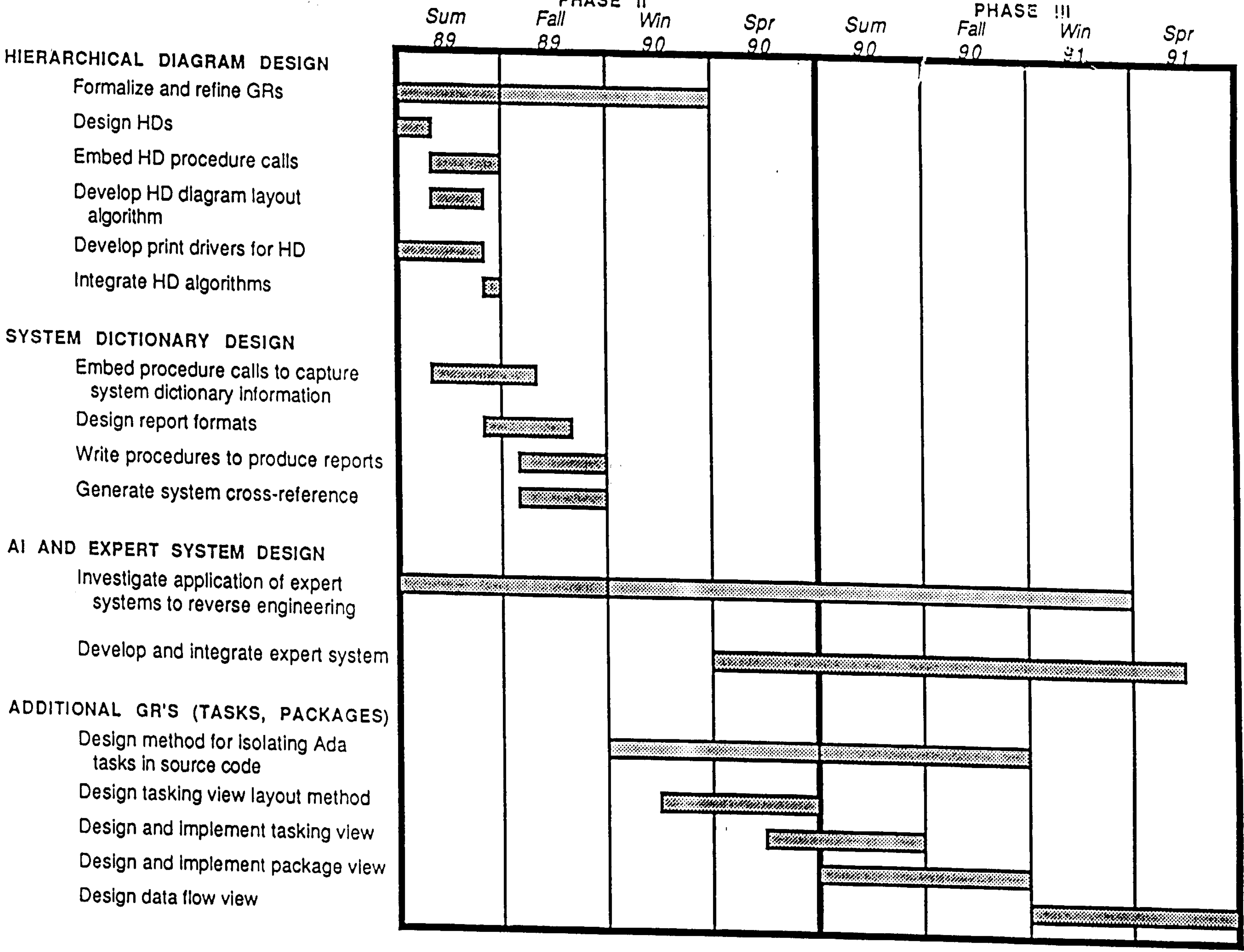
5. Investigate the application of artificial intelligence (AI) and expert systems to this graphically-oriented methodology.

The use of expert systems and rule-based systems will be investigated as an approach to analysis of Ada software. In particular, AI-assisted layout of the graphical representations described above will be investigated.

TASK 2: PROPOSED RESEARCH SCHEDULE

The following Gantt chart provides the sequence of activities to be accomplished during Phase 2 and Phase 3 of this project. Details for the Phase 2 activities were presented above. Phase 3 activities will be developed in greater detail as Phase 2 progresses.

REVERSE ENGINEERING TOOLS FOR ADA SOFTWARE
PROPOSED RESEARCH ACTIVITIES



ORIGINAL PAGE IS
OF POOR QUALITY