QUEST/Ada

QUERY UTILITY ENVIRONMENT FOR SOFTWARE TESTING OF ADA

The Development of a Program Analysis Environment for Ada

Contract Number NASA-NCC8-14

First Six-Month 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) 826-4330
dbrown@AUDUCVAX.bitnet

November 30, 1988

TABLE OF CONTENTS

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
Orville R. Weyrich
Michael P. Woods

UNDERGRADUATE TECHNICAL ASSISTANTS

Todd E. Blevins
Edward Swan

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 it predecessor in the following regard: (1) the parser/scanner mechanism will be obtained from a formal parser/scanner generator such as YACC, LALR 3.0, or BISON, (2) the test data generator will be 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 user interface.

The six-month 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. This is followed by a major 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 is also included in the appendix which gives statistical evidence of the validity of the test case generation approach which is being integrated into QUEST/Ada.

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 are being 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 subset to be addressed by the prototype is defined. This is followed by the definition of parser/scanner requirements, which contains an example module instrumented by an early prototype. Section 7 presents an early view of the rule-based test data generator, after which the plan for the remainder of the project is given. Finally, the high level IORL description of QUEST/Ada is given in the Appendix.

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

allowing "...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 will address 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 will use 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.

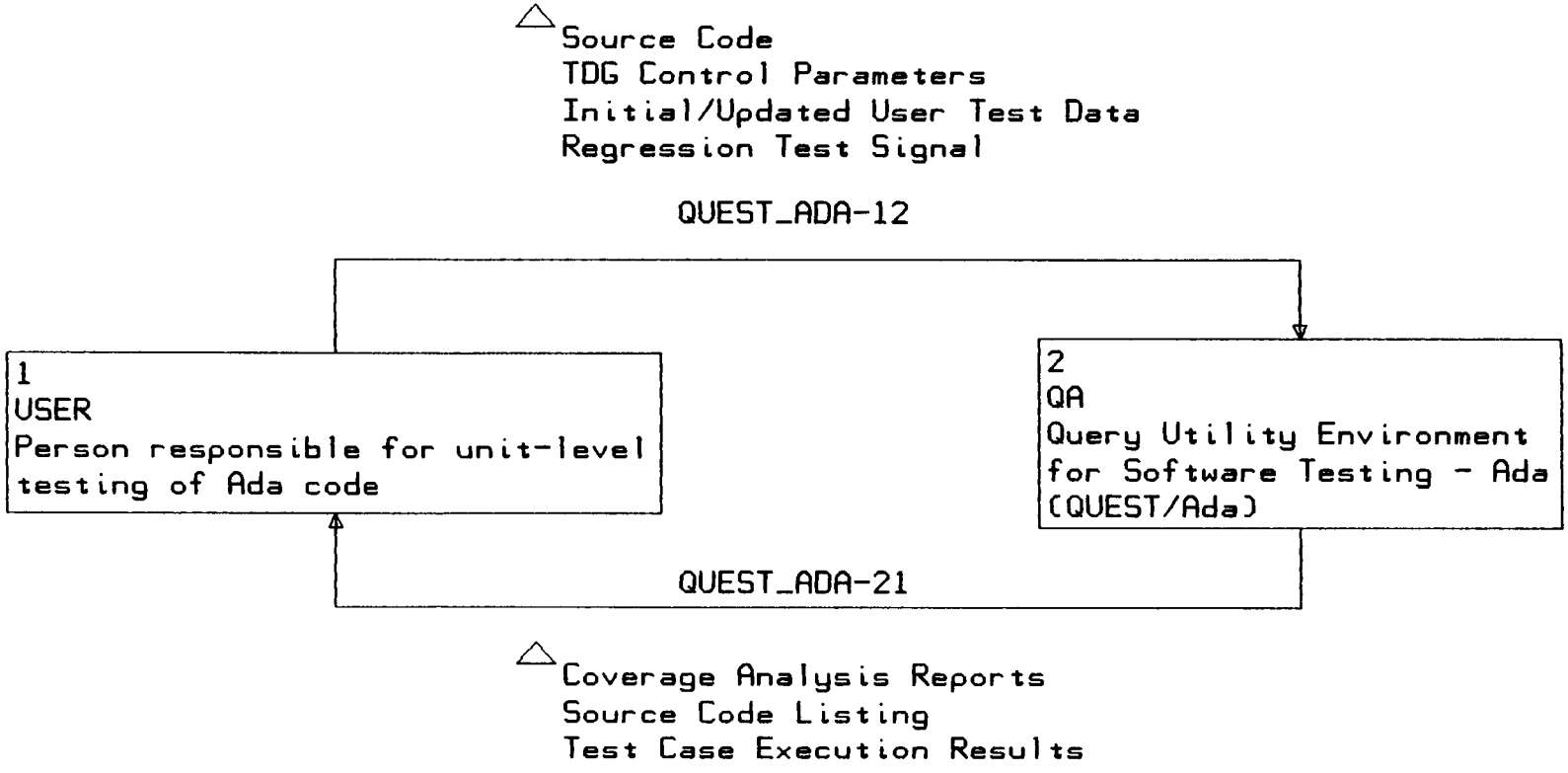


FIGURE 1. - HIGHEST LEVEL SCHEMATIC BLOCK DIAGRAM

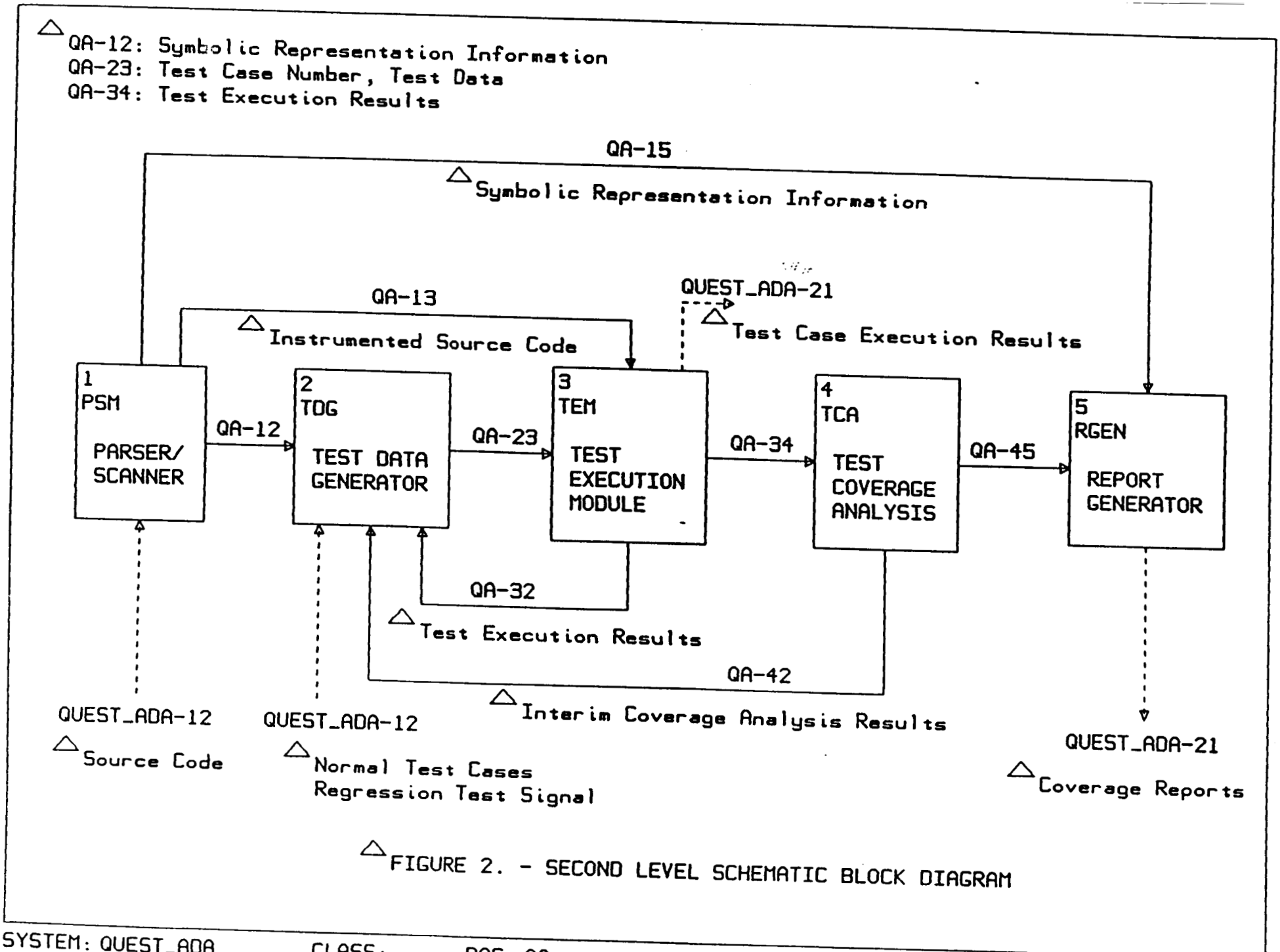


FIGURE 2. - SECOND LEVEL SCHEMATIC BLOCK DIAGRAM

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

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 instruments the source code by inserting probes and augmenting it 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

QUEST_ADA-21, SOURCE CODE LISTING
TO: USER

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 generator 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-32 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.

Also shown in Figure 2 is the potential use by the TDG of subcomponents of the Test Coverage Analysis (TCA) (Component 4 in Figure 2). It is currently envisioned that the same types of analysis performed by the TCA will be used in the TDG. The extent of interaction between these two modules will be resolved during the detailed design.

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

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-32, TEST EXECUTION RESULTS FROM: TEST EXECUTION MODULE
	QA-42, COVERAGE ANALYSIS RESULTS FROM: TEST COVERAGE ANALYSIS
OUTPUTS:	QA-23, TEST CASES TO: TEST EXECUTION MODULE
	1. TEST CASE NUMBER
	2. TEST DATA

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 Report Generator.

A more complex problem is posed by the requirements of the TDG. Information from TEM must enable TDG to fire additional actions from its rule base. Thus, the information must be translated to a set of assertions either by TEM, TCA or TDG itself. These responsibilities will be more specifically assigned as the rule base design matures.

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-32, TEST EXECUTION 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-34, TEST EXECUTION RESULTS
TO: TEST COVERAGE ANALYZER

1. TEST CASE NUMBER
2. DECISION NUMBER
3. LIST OF CONDITION RESULTS

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 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 might be as feedback to the TDG to automatically generate improved test cases. However, the degree of interaction between these two modules has not yet been resolved.

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 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

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

Table 6. REPORT GENERATOR INTERFACES

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

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. Definition of Ada Subset

The formidable task of constructing a working prototype of an automated testing environment during a one-year period requires a limitation on the scope of the project. Since the goal of the prototype is to automatically generate 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.

In the area of module input variables or parameters, an attempt will be made to handle all scaler types and subtypes. These include integer, float, real, character, Boolean, and enumerated types. The test environment will also be designed to generate data for arrays and records (composite types) of these simple types. No access types will be handled by the prototype. If it is found to be infeasible to generate a prototype with these capabilities, then a representative subset of types will be selected. However, no decision has been made to eliminate other than access types at this point. Further consideration in the current design will be made in order to determine methods for including access types during Phase 2.

For programs which obtain inputs from files, the same restrictions will apply. Records with discriminants and linked components will be deferred to the next prototype version. These limitations are necessary because they require knowledge about the file and data structures that cannot be obtained directly from the code being tested. During Phase 2, formal input specifications will be developed to handle complex data

structures and files. Consideration will be given in this phase to establish the basis for these specifications.

The initial prototype will generate test cases for multi-tasking 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 will be given during the prototype design and development to establish approaches for handling concurrency. However, the actual prototyping of these approaches will be deferred until Phase 2.

6. Preliminary Analysis of Parser/Scanner Requirements

6.1 General Parser/Scanner Requirements

The parser/scanner module is responsible for instrumenting the Ada source code, building the data structures required by the rest of the QUEST system, creating a listing of the source code for use by the tester, and surrounding the module under test with an execution driver or test harness. Information contained in the data structure must identify the control constructs, global variables referenced (i.e., altered) within the module, and parameters input to the module.

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 is used as an indication of test case quality for directing the generation of new test data.

The data structures built by the parser will 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 will show the embedded instrumentation code added by the parser. Unique identification numbers will be assigned to each decision, condition, and code segment in the original code listing.

The last requirement of the parser/scanner module is the creation of a driver module to execute the program under test. This driver reads data from a file created by the test data generator and feeds this data to the instrumented object code. This process occurs repeatedly until the current set of test data is exhausted.

Two parser/scanner generator packages, LALR 3.0 and BISON, were evaluated for use in producing the instrumentation capabilities. 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 in the following subsection.

6.2 Example Module Instrumentation

In order to test BISON as a parser/scanner generator some simple examples were run. This very early prototyping was necessary in order to determine if there were any obstacles to using this tool for generating the instrumentation. Listing 1 presents the first example which was tried. Note that it contains two "if" statements. Listing 2 shows how these were replaced by the subroutine calls d0 and d1 respectively. This replacement was performed automatically by the parser/scanner. Note that line reference numbers were also added for further use by the report generators.

While this is a very simple example, it demonstrates the concept, and it represents progress far ahead of what was expected at this point. Given that BISON is proven in this regard, the second six months of this phase of the project can extend the parser/scanner capabilities to a set of representative transfer statements within Ada.

Listing 1. Example Ada Module

```
        --*********************************************************
        -- ADA EXAMPLE PROGRAM:  Max3 - This program computes
        --                              the maximum integers,
        --                              and prints the result
        --                              terminal screen
        --*********************************************************

with TEXT_IO; use TEXT_IO

procedure MAX3 is

     package INT_IO is new INTEGER_IO(INTEGER);
     use INT_IO;

     I, J, K, L:  INTEGER;

begin

     -- input the three values from the screen
     GET(I); GET(J); GET(K);

     -- compute the maximum of I and J
     if I > J then
        L := I;
     else
        L := J;
     end if;

     -- compute the maximum of I, J, and L
     if L < K then
        L := K;
     end if;
     -- print out the answer
     NEW_LINE;
     PUT(" The largest is: ");
     PUT(L);
     NEW_LINE;

end MAX3;
```

Listing 2. Instrumented Example Ada Module

```
[1]       --*******************************************************
[2]       -- ADA EXAMPLE PROGRAM:  Max3 - This program computes
[3]       --                              the maximum integers,
[4]       --                              and prints the result
[5]       --                              terminal screen
[6]       --*******************************************************
[7]
[8]       with TEXT_IO; use TEXT_IO
[9]
[10]      procedure MAX3 is
[11]
[12]           package INT_IO is new INTEGER_IO(INTEGER;
[13]           use INT_IO;
[14]
[15]           I, J, K, L: INTEGER;
[16]
[17]      begin
[18]
[19]           -- input the three values from the screen
[20]           GET(I); GET(J); GET(K);
[21]
[22]           -- compute the maximum of I and J
[23]           if d0(  I > J ) then
[24]              L := I;
[25]           else
[26]              L := J;
[27]           end if;
[28]
[29]           -- compute the maximum of I, J, and L
[30]           if d1(  L < K   ) then
[31]              L := K;
[32]           end if;
[33]           -- print out the answer
[34]           NEW_LINE;
[35]           PUT(" The largest is: ");
[36]           PUT(L);
[37]           NEW_LINE;
[38]
[39]      end MAX3;
```

7.0 Preliminary Analysis of the Test Data Generator (TDG)

The objective of the test data generator is to generate a set of test data that will cover as many conditional branches in a program as possible. Typical conditional branches are implemented in IF-THEN and CASE statements. At this point attention will be focused on the coverage of the IF-THEN branches. An IF-THEN statement can be expressed as

IF cond THEN f1 ELSE f2

The logical expression, cond, determines the branch of the next execution. In order to cover all branches of the statement, i.e. f1 and f2, a set of test data should provide conditions such that cond would be true in some cases and false in other cases. A necessary test data set may be defined as a pair of test inputs where one provides truth value for cond and the other provides false value. Cond can be further defined as

cond: exp1 rel exp2

Exp1 and exp2 can be any arithmetic expressions. After evaluation, each of the expressions will yield a numerical value. Rel is either =, <, =<, >, >=, or <>. The evaluation of cond would yield truth or false value. No matter what the rel is, the inclusion of all the following three cases would guarantee that a test data set covers both the true and the false statement of an IF-THEN statement:

1. exp1 + e = exp2
2. exp1 = exp2
3. exp1 - e = exp2

Here, e is defined as a small positive number. The basic

objective is to generate a test data set that covers both sides of the truth/false boundary of cond. This will be the guideline for the test data generation for this phase of the project. This general approach has been tested in a very rudimentary form, and the results have been summarized in a paper given in Appendix B. The approach showed great promise in automatically generating coverage far superior to that obtained by random test case generation.

7.1 Cases of Arithmetic Expressions

In order to generate test data that will cover the three cases listed above, the structures of the arithmetic expressions, i.e. exp1 and exp2, must first be recognized. The following list shows the structures that will be studied:

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

The reason for restricting consideration to these relatively simple structures is that the condition boundaries of the expressions can be found through simple arithmetic computations. For more complicated expression structures, mathematical subroutines can be used to find the boundaries.

One further assumption required to initiate prototype design is that the variables appearing in exp1 and exp2 are input

variables. This means that, aside from the arithmetic operators, the components of exp1 or exp2 must be either a constant or an input variable.

7.2 Heuristics For Finding the Condition Boundaries

The computation for finding the condition boundaries can be greatly simplified by rearranging the logical expression, cond, in the IF-THEN statement. The following two rules will be used for this purpose:

Rule 1

```
If      exp1 does not contain variables
then    (1) swap exp1 and exp2
        (2) adjust rel
```

Rule 2

```
If      exp1 contains constants
then    move all possible constants to exp2
```

These rules simplify exp1 such that it contains at least one variable and no constants. This arrangement reduces the number of combination cases between exp1 and exp2. For example, given a condition

```
3  =<  5 * X + 4

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

By applying Rule 1, it becomes

```
5 * X + 4  >=   3
```

By applying Rule 2, it becomes

```
X  >=   -0.2
```

From this simplification process, the condition boundary can be found without going through other computations. For the above example, three test data points can be generated for X. They are X = -0.2 + e, X = -0.2, and X = -0.2 - e.

Now, we will study all possible combinations of exp1 and exp2. Under each combination, a set of test data is suggested. The generalized cases include:

1. exp1: X exp2: C1

   boundary: X = C1

   test data : 1. X = C1 - e
               2. X = C1
               3. X = C1 + e

   Note: e = (upper-bound - lower-bound) of X / 100

2. exp1: X exp2: Y

   boundary: X = Y

   assign y_1 = (upper-bound + lower-bound) of Y / 2

          Y = y_1

   test data: 1. X = y_1 + e , Y = y_1
              2. X = y_1 , Y = y_1
              3. X = y_1 - e , Y = y_1

   Note: Since the goal is to generate a set of test data that would cover both sides of the boundary, it does not matter which portion of the boundary the test data resides. The choice made here is to let Y be at the middle point of its range.

3. exp1: X exp2: Y + C1.

   boundary: X = Y

   assign y_1 = (upper-bound + lower-bound) of Y / 2
          Y = y_1

   test data: 1. X = y_1 + C1 + e , Y = y_1
              2. X = y_1 + C1 , Y = y_1

3. $X = y_1 + C1 - e$, $Y = y_1$

4. exp1: X exp2: Y * C1 (or Y / C1)

boundary: X = Y * C1

assign y_1 = (upper-bound + lower-bound) of Y / 2
Y = y_1

test data: 1. $X = y_1 * C1 + e$, $Y = y_1$
2. $X = y_1 * C1$, $Y = y_1$
3. $X = y_1 * C1 - e$, $Y = y_1$

5. exp1: X exp2: C1 * X + C2 * Y + C3

simplification steps:

X	rel	C1 * X + C2 * Y + C3
(1-C1) * X	rel	C2 * Y + C3
X	rel'	C4 * Y + C5

boundary: X = C4 * Y + C5

assign y_1 = (upper-bound + lower-bound) of Y / 2
Y = y_1

test data: 1. $X = C4 * y_1 + C5 + e$, $Y = y_1$
2. $X = C4 * y_1 + C5$, $Y = y_1$
3. $X = C4 * y_1 + C5 - e$, $Y = y_1$

6. exp1: X exp2: C1 * X * Y + C2

assign y_1 = (upper-bound + lower-bound) of Y / 2
Y = y_1

boundary: X = C1 * y_1 * X + C2 = C3 * X + C2

simplification steps:

X	rel	C3 * X + C2
(1-C3) * X	rel	C2
X	rel'	C4

test data: 1. X = C4 + e , Y = y_1
2. X = C4 , Y = y_1
3. X = C4 - e , Y = y_1

7. exp1: C1 * X + C2 * Y exp2: C3 * X + C4 * Y + C5

simplification step:

C1 * X + C2 * Y	rel	C3 * X + C4 * Y + C5
(C1 - C3) * X	rel	(C4 - C2) * Y + C5
X	rel'	C6 * Y + C7

the condition then becomes a case of (6).

8. exp1: C1 * X + C2 * Y exp2: C3 * X * Y + C4

assign y_1 = (upper-bound + lower-bound) of Y / 2
Y = y_1

simplification steps:

C1 * X + C2 * y_1	rel	C3 * X * y_1 + C4
C1 * X + C5	rel	C3 * X + C4
C1 * X	rel	C3 * X + C6
(C1 - C3) * X	rel	C6
X	rel'	C7

the problem then becomes a case of (1).

9. exp1: X * Y exp2: C1

boundary: X * Y = C1

assign y_1 = (upper-bound + lower-bound) of Y / 2
Y = y_1

simplification steps:

X * y_1	rel	C1
X	rel'	C2

test data: 1. X = C2 + e , Y = y_1
2. X = C2 , Y = y_1
3. X = C2 - e , Y = y_1

10. exp1: X exp2: C1 * Y / X + C2

assign y_1 = (upper-bound + lower-bound) of Y / 2

$Y = y_1$

simplification steps:

X rel $C1 * y_1 / X + C2$

X rel $C3 / X + C2$

$X^2 - C2 * X - C3$ rel 0

assign $x_1 = (C2 + SQRT (C2^2 + 4 * C3)) / 2$

test data: 1. $X = x_1 + e$, $Y = y_1$
2. $X = x_1$, $Y = y_1$
3. $X = x_1 - e$, $Y = y_1$

Note: Since the goal is to cover both sides of the boundary, three data points will be sufficient.

7.3 Structural Methods

The Test Data Generator (TDG) uses structural methods to automatically generate a series of test packets to fully exercise the module under test. The initial prototype will attempt to obtain 100% condition/decision coverage, although the concept could be extended to any type of coverage metric.

The traditional technique for generating test data using structural or syntax-based methods is to: (1) determine the desired path through a program, (2) use a form of symbolic execution to obtain a predicate for that path, and (3) solve the path predicate in terms of the program's input variables. By executing the program with the calculated input variables, the desired path will be executed. QUEST/Fortran used this method and determined that path predicate solution was too complex to be universally effective.

The technique used by QUEST/Ada differs considerably from

the above-mentioned technique. The new system attempts to determine the relationships between the input variables and the decisions involved in the program's flow control constructs. Most of these decisions may be described by the following grammar:

```
decision::          condition
                    | condition logical_op condition
condition::         expression rel_op expression
logical_op::        and | or | xor
rel_op::            =  |  /=  |  <  |  <=  |  >  |  >=
```

In other words, decisions consist of conditions separated by logical operators and conditions consist of expressions separated by relational operators. Each of these expressions may be considered to be a function of the program's inputs. If a particular function were known, then it would be a trivial matter to calculate the input parameters necessary to force the condition to be true or false. Although the exact function cannot be determined without symbolic execution, information about the function may be obtained by inserting probes in the source code so that the value of the expression may be evaluated and saved at run-time. Then, by observing the response of the expression for various input parameters, the relationship between the inputs and the expression may be identified. Although situations occur where the function is too complex and therefore impossible to identify by looking only at the inputs and outputs, by confining the domain of interest to those situations where the expression on the left-hand side of the condition is almost equal to the expression on the right-hand side, an approximation of the

function may be determined. The reason for making this zero-crossing point the domain of interest is two-fold: (1) the truth value of the condition changes at this point, and (2) many errors in control flow logic are uncovered with test data that force the expressions into this domain [HOW87].

The Test Data Generator (TDG) operates by looking at the results of the previously run tests and determining those decisions that have only had one side of their truth value covered. The TDG then examines each of the conditions comprising the decision. The test data is analyzed and those sets of data that force the left-hand side of a condition close to the right-hand side are then slightly modified in an attempt to drive the condition to its other truth value. The justification behind this is found in Prather [PRA87] and is summarized here. If a particular condition, Cn, is reached, then all the preceding conditions, C1 through Cn-1, along the path have also been satisfied. In order to drive the target condition, Cn, to its' other truth value, all of the preceding 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 TDG.

The TDG uses a variety of methods for slightly modifying the test sets. If it can be determined that an expression is always

increasing (or decreasing) with respect to an input variable over the domain of interest, then Newton's method is used to calculate a new test set. Other means of generating new test data include incrementing (and decrementing) by a constant, incrementing (and decrementing) by a percentage, and generating a random number for one of the parameters. As the rule base develops other methods will also be considered.

Since the TDG generates data only for those decisions that have had one of their truth values covered, there must be a way to initialize it. This may be accomplished by user-defined test sets or by the generation of random data. Provisions have also been made to allow the user to enter designed test cases at any time during the testing process. Additionally, the user may hold one or more of the inputs constant while the TDG generates data for the other inputs.

7.4 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 test cases being supplied from somewhere. While they will view this information as coming

through the same interfaces as data actually generated by TDG (and hence will respond exactly the same), in reality the original set of data supplied to TEM will either be user supplied or randomly generated. 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 prior to automatically generating test cases.

As far as the interface mechanism is concerned, the user will have placed these test cases in a file prior to the initiation of module testing. These will be passed through TCA to TEM for the first round of tests. This will effectively prime the pump to enable TEM and TCA to return coverage and execution information which will drive the TDG. At this point TDG will use this information to generate another packet of test cases which will be added to the file of test cases and marked as being TDG rather than user produced.

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 TDA 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. Preliminary 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 began to evolve it tended to contribute 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 should be

regarded as a working document. It is expected to continue to evolve throughout the remainder of this phase of the project. It will also provide the basis for the user manual for the QUEST/Ada system.

Figure 3 gives an overview of the user interface as it interacts with the four components of the system (compare with the IORL SBD, document: QUEST). The QUEST Main Menu, given in Menu 0 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 3. Each of these will be described in a separate subsection below.

8.1 System Definition Menu

When this selection is chosen from the QUEST Main Menu, Menu 1 will appear. This menu enables the user to create and delete a "system" within QUEST. In this context, a "system" is a complete functional collection of Ada source code files. That is, all modules necessary for executing any of the units to be tested must be included in the system at this time. We will refer to this system below as the system under test or SUT.

When the System Definition screen is initially displayed, the directory of the current default pathname (initialized to "*.ADA" by QUEST) is visible in the text window. A user may then select any of the files displayed by highlighting them with the arrow keys and pressing "Return." The number of files selected for inclusion in the QUEST system is constrained only by the

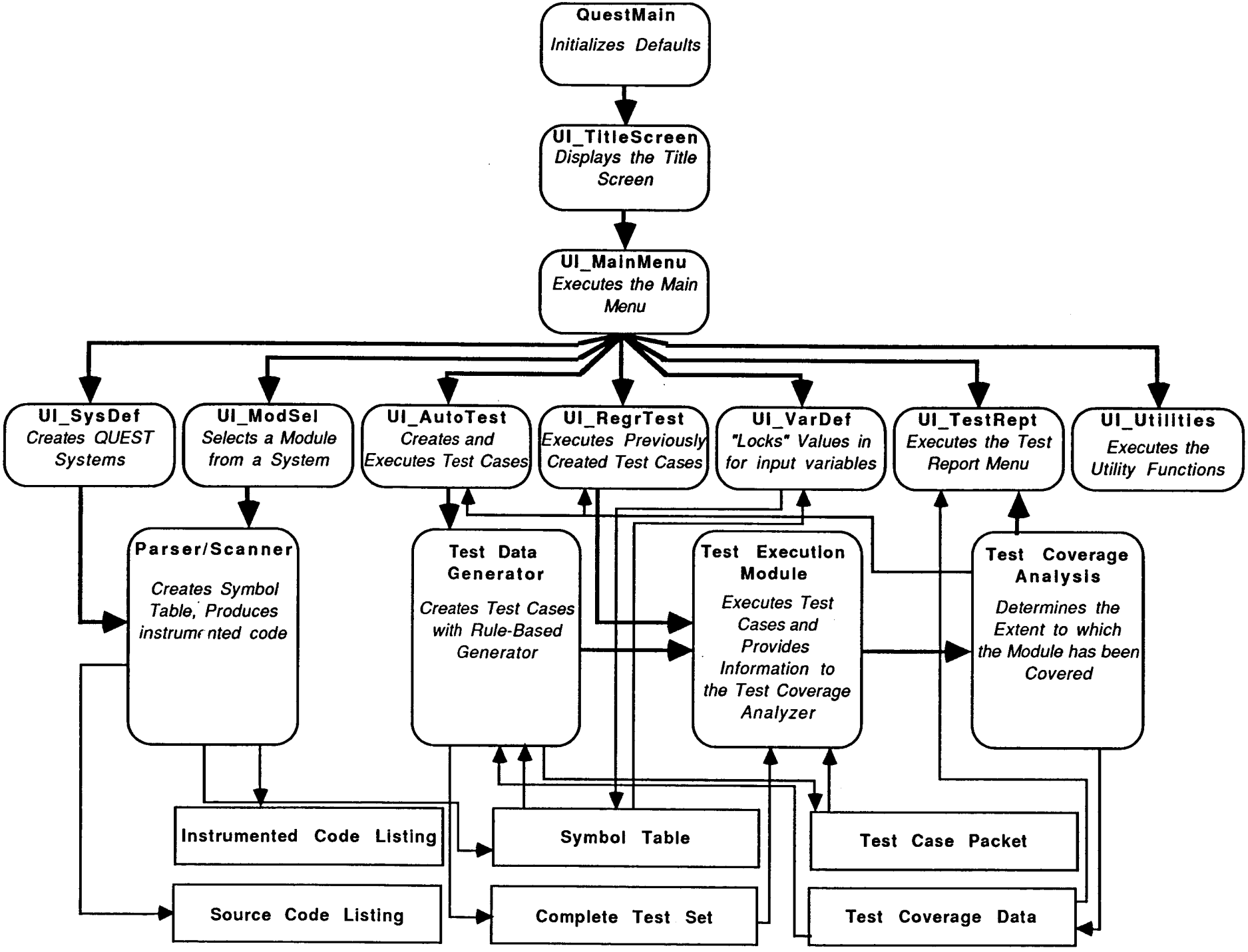


FIGURE 3: Overview of the User Interface

QUEST Main Menu

1 System Definition

2 Module Selection

3 Automatic Testing

4 Regression Testing

5 Variable Definition

6 Test Result Reports

7 Utilities

PF1 - Help

PF4 - Exit

Current Module:

Menu 0. QUEST Main Menu

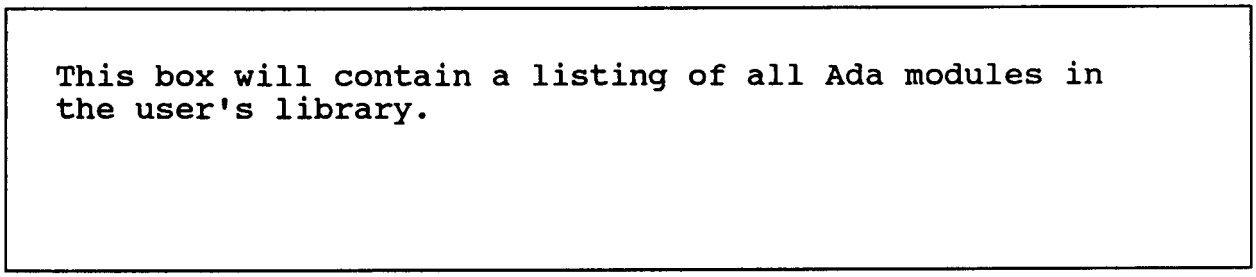
memory limits of the computer. When the user has selected all of the files to be included in the QUEST system, pressing "PF2" will create that system and prompt the user for a system name. As with all QUEST menus and screens, "PF1" displays the help screen, and "PF4" returns to the main menu.

8.2 Module Selection Menu

When this selection is chosen from the QUEST Main Menu, Menu 2 will appear. This menu allows the user to select the module under test (MUT). Note that it is left to the QUEST user to

QUEST

System Definition Menu



Select files and press return

PF1 - Help

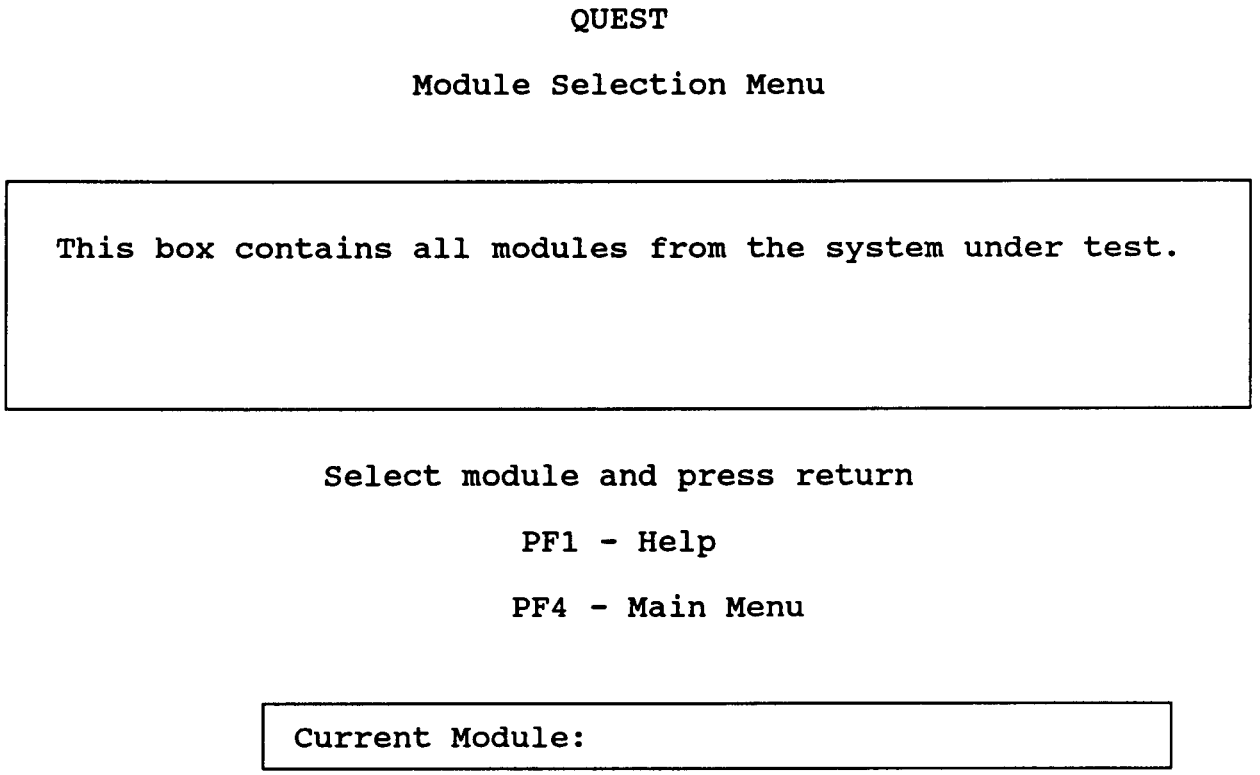
PF2 - Create System

PF3 - Delete System

PF4 - Main Menu

Menu 1. QUEST System Definition Menu

insure that all modules necessary to the execution of the MUT are included in SUT. If a module necessary to the execution of the MUT is not in the SUT, the parser/scanner will return an error. When the Module Selection menu is initially displayed, the name of the current SUT and all modules included in that system are



Menu 2. QUEST Module Selection Menu

displayed in the text window. The user can select a module to test by highlighting it with the arrow keys and pressing "Return." Unlike the system definition screen, only one module at a time may be selected for testing. When the user has selected a module, pressing "PF4" returns to the main menu.

8.3 Automatic Testing Menu

When this selection is chosen from the QUEST Main Menu, Menu 3 will appear. This menu monitors the generation and execution of test cases. When the Automatic Testing screen is initially displayed, the user is prompted for a maximum number of test

QUEST

Automatic Testing Menu

Maximum Number of Test Packets:
Packets Created:
Tests Created:

Last Test Executed:
Coverage Achieved:
Decision:
Condition:

User Defined Variables:

PF1 - Help

PF2 - Begin Testing

PF3 - Halt Testing

PF4 - Main Menu

Current Module:

Menu 3. Automatic Testing Menu

packets to create. Each test packet generated will contain a certain number of test cases to be executed by the TEM. QUEST

initializes the number of test cases per packet to 50, but users may change the number using the Utilities menu.

After the user has specified the maximum number of packets to create, "PF2" initiates the generation and execution of test cases. As the tests are created, the number of packets and the number of test cases created is reported on the Automatic Testing screen. After a complete test packet has been generated, the TEM begins executing tests. The last test executed and the coverage achieved to that point are reported to the Automatic Testing screen by the TCA. The input variables (i.e., those variables for which values can be generated by the TDG) whose values have been set explicitly by the user are also reported on the Automatic Testing screen. The user may request a halt to the test generation/execution at any time. However, test data generation and execution will only stop upon completion of a test packet. When the user requests a halt, a message that the request was acknowledged is displayed on the screen, and test generation/execution stops as soon as possible. If test execution completes successfully, a message to that effect is displayed and the user can press "PF4" to return to the main menu.

8.4 QUEST Regression Test Menu

When this selection is chosen from the QUEST Main Menu, Menu 4 will appear. This menu enables files of previously performed tests to be executed again automatically. This is essential

after any program modification to assure that errors have not been introduced during debugging. The data reported to the Regression Testing screen is identical in form and meaning to the information reported to the Automatic Testing screen, except that data which pertains to the generation of test cases. The "PF2" and "PF3" keys also work in the same way as those on the Automatic Testing screen.

QUEST

Regression Testing Menu

Tests on File:

Last Test Executed	
Coverage Achieved	
Decision	
Condition	

PF1 - Help

PF2 - Begin Testing

PF3 - Halt Testing

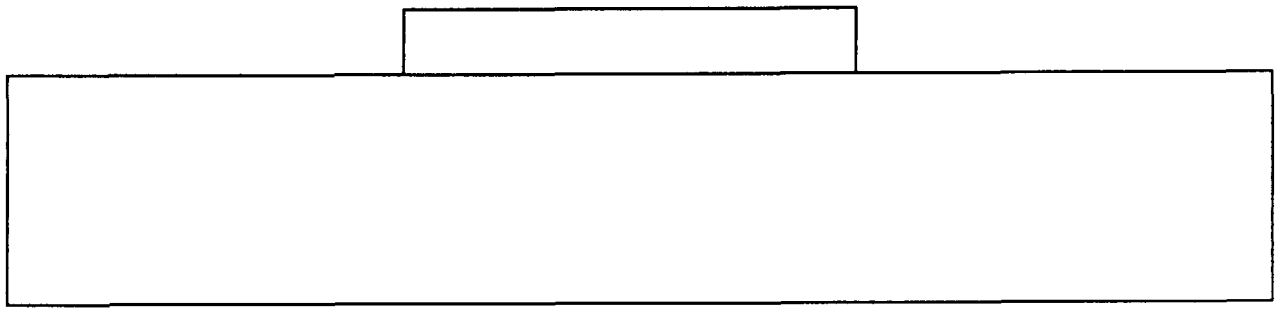
PF4 - Main Menu

Current Module:

Menu 4. Regression Test Menu

8.5 QUEST Variable Definition Menu

When this selection is chosen from the QUEST Main Menu, menu 5 will appear. This menu enables users to fix values for any or all of the input variables of the MUT. This process is referred to as "locking" the variables, as user definition of values prevents the TDG 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 text window. 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. The user may select a variable for definition 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. The user can then enter a new value for that variable in the "New Value" field.



Menu 5. Variable Definition Menu

8.6 Testing Result Reports Menu

When this selection is chosen from the QUEST Main Menu, Menu 6 will appear. This menu enables the selection of reports patterned after those generated in QUEST/FORTRAN [BRO87].

8.7 QUEST Utilities Menu

When this selection is chosen from the QUEST Main Menu, Menu 7 will appear. These are miscellaneous utilities necessary for the functions of QUEST but not logically falling within the other

routine QUEST functions.

8.8 Summary of User Interface Design

In the original plan it was not envisioned that the user interface would be to this state of design at this time. However, given the user interface of QUEST/FORTRAN along with the solidification of the new test-case-generation approach, the preliminary design of the user interface could proceed. The documentation given above will form the basis for an early user interface which will facilitate the remainder of the design and

QUEST

Testing Result Reports Menu

1 Test Coverage Report

2 Cumulative Coverage Report

3 Regression Test Report

Select menu option and press return:

PF1 - Help Menu

PF4 - Main Menu

Current Module:

Menu 6. Test Results Reports Menu

development of the other component prototypes. For this reason this portion of the design/development is being allowed to lead the others. Recognize that many modifications of the user interface design are expected. The documentation in this section will be modified and heavily augmented during prototype development to form the user manual.

9. Detailed Plan for Project

Chart 1 is a Gantt chart which shows the project activities for Phase 1 and their expected duration. All activities shown to

QUEST

Utilities Menu

Execute Single Test Case

Regression Test Set Default:
Minimal Test Set
Complete Test Set

Delete System

PF1 - Help
PF4 - Main Menu

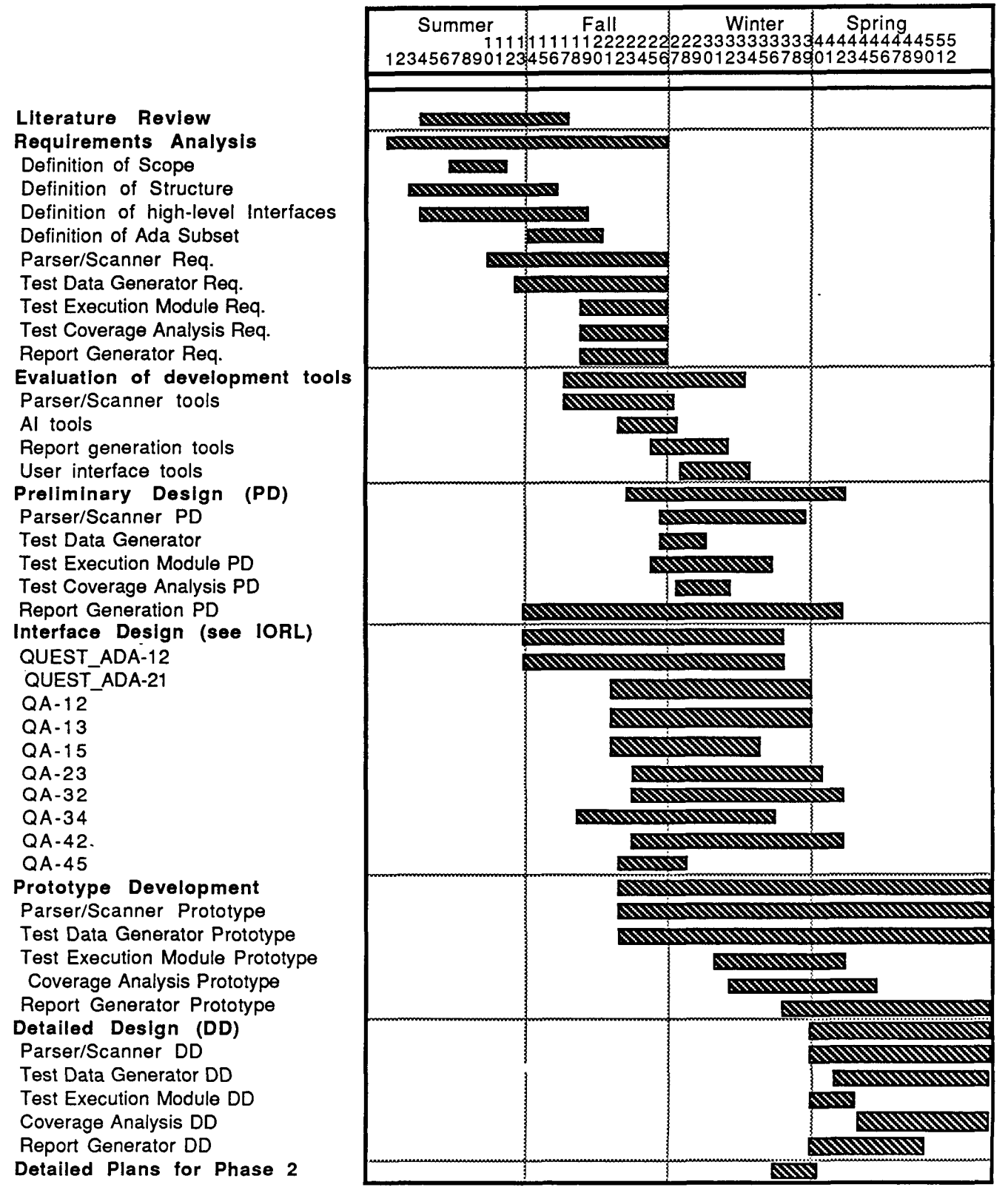
Current Module:

Menu 7. QUEST Utility Menu

be completed by week 26 have been completed. The remainder are either in progress or are yet to be initiated. In summary, the following activities have been completed: (1) literature review, (2) requirements analysis (all subactivities), and (3) evaluation/selection of development tools, with the exception of the report generation and user interface tools. The following activities have been initiated and are still in progress: (1) preliminary design (all subactivities), (2) interface design (all subactivities), and (3) prototype development for the parser/scanner and the test data generator. The remainder of the activities, including prototype development for the remaining components and all detailed design activities have not yet been initiated. These will be initiated at the start times indicated by the Gantt chart.

The plans given above are for the first year, which is the first phase of a three-phase project to design and develop a prototype environment to facilitate Ada code testing. Detailed plans for Phase 2 will be made as indicated in the Gantt chart. These have been deferred to take advantage of knowledge gained during Phase 1. At this point the following broad requirements statements can be made with regard to the continuation of this project into Phase 2: (1) Refinements will be required in order to improve the efficiency of QUEST/Ada and make the prototype more generally applicable, (2) Concurrency constructs will require that the dimensions of time and sequence be considered (the prototype designed under Phase 1 has not included such

Chart 1. Gantt Chart for Project Planning for Phase 1



consideration), and (3) A major effort will be required to extend the current prototype to the broad range of types which Ada supports, especially access types. Detailed plans for these activities will be discussed with NASA technical management as well as Ada practitioners as Phase 1 continues.

Plans for Phase 3 are still quite tentative. However, it appears that this phase will be required to turn the prototype environment into a working production quality system useful for field evaluation and actual Ada system code testing. The original proposal coupled the university contractor with a private subcontractor for the major system development activities of QUEST/Ada. As the prototypes continue to be developed and tested, this approach will be evaluated.

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.

---

*Reference not discussed in Section 2.

[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.

[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.

[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.

[WOO80] 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. 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 data

*For details obtain the IORL Reference Manual, Teledyne Brown Engineering, Inc., 1984.

flow (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 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.

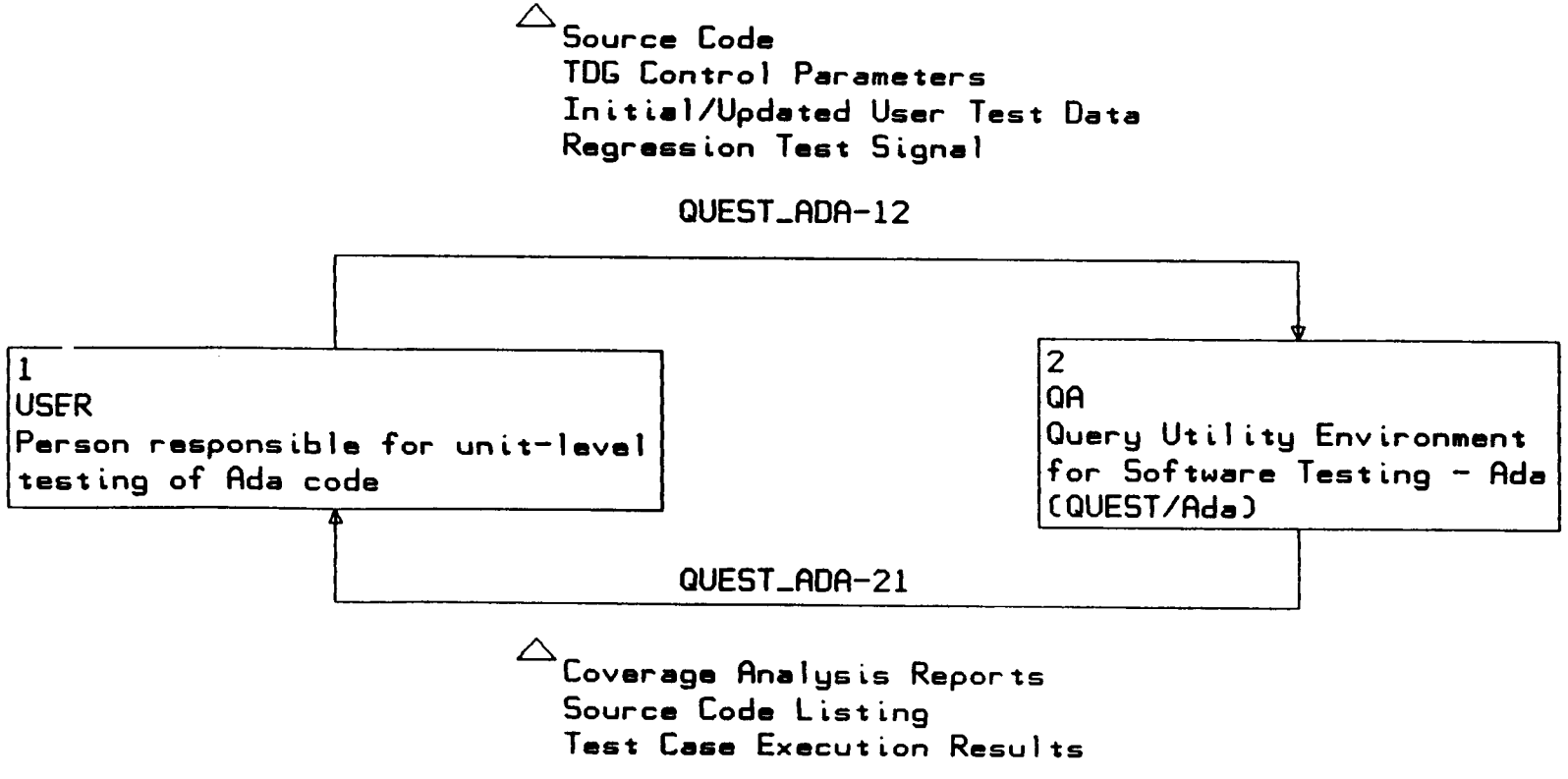


FIGURE 1. - HIGHEST LEVEL SCHEMATIC BLOCK DIAGRAM

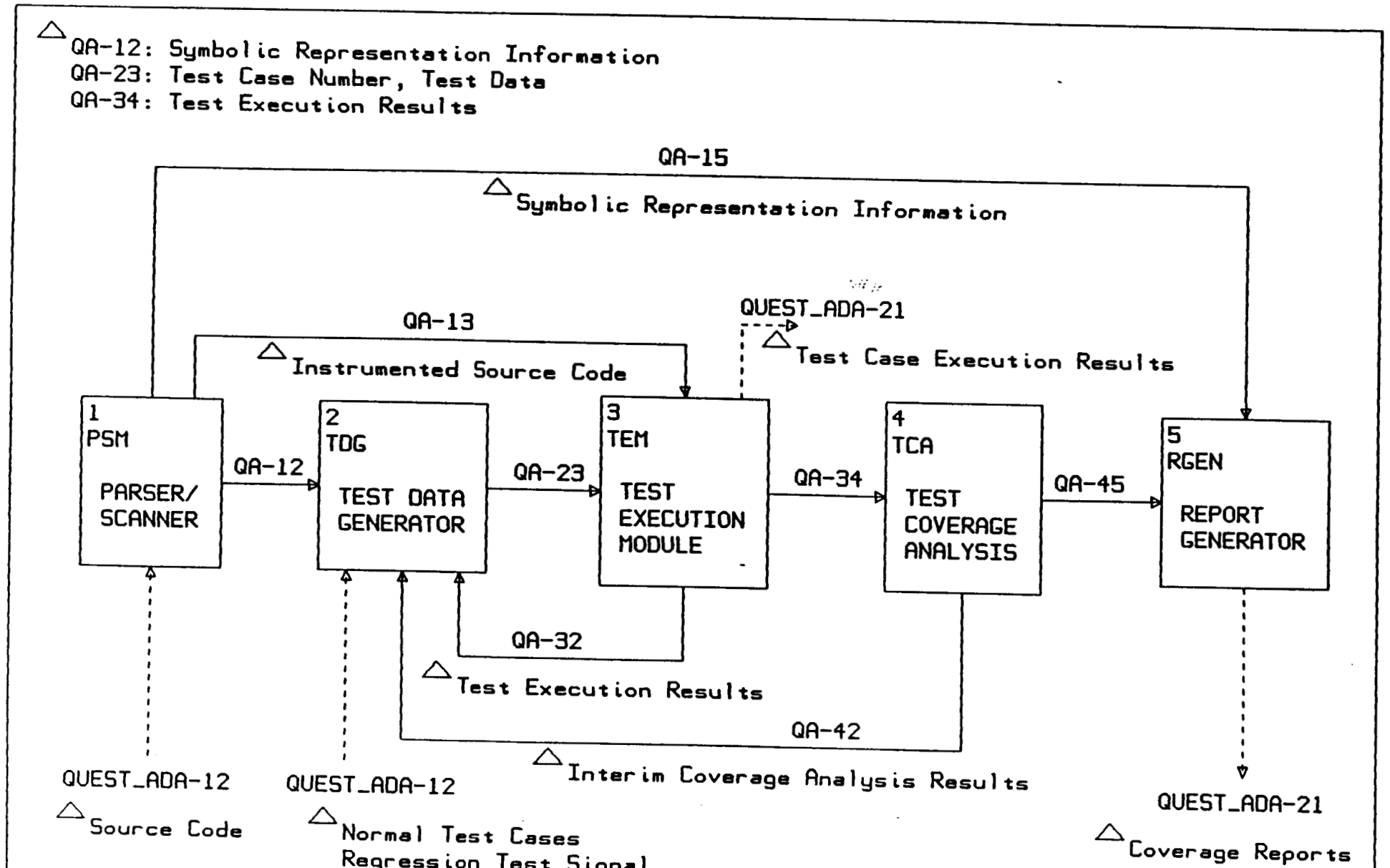
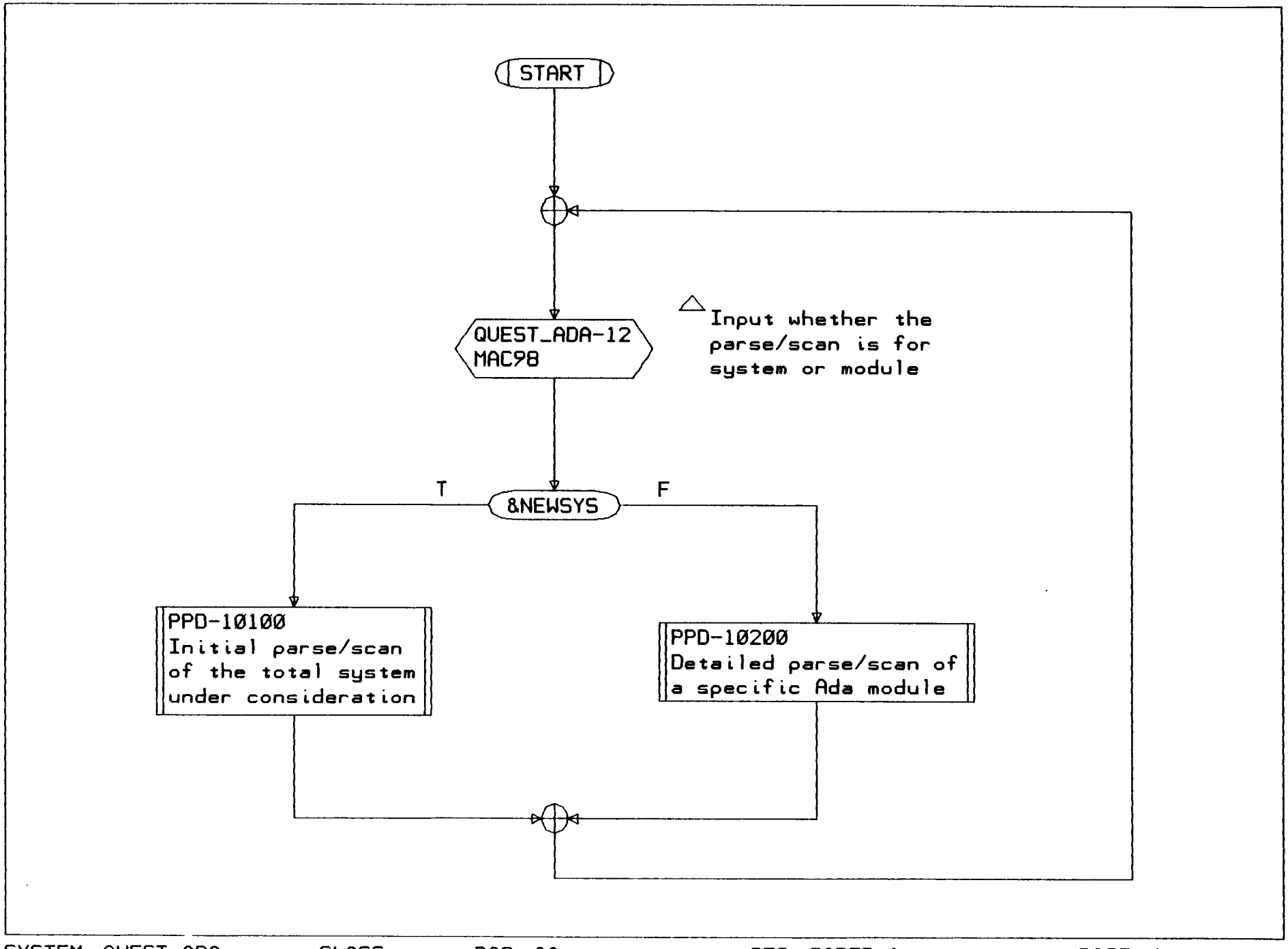
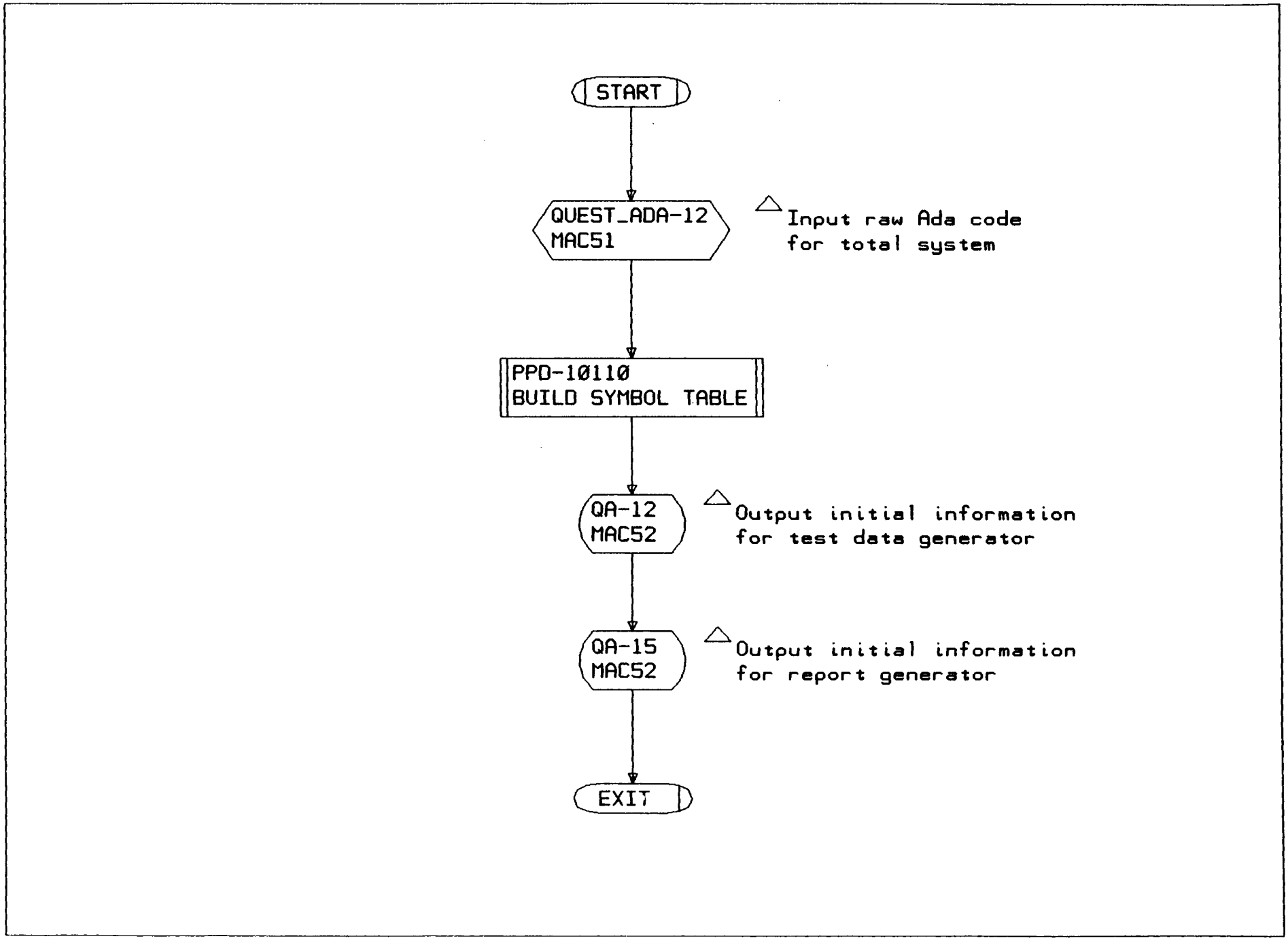
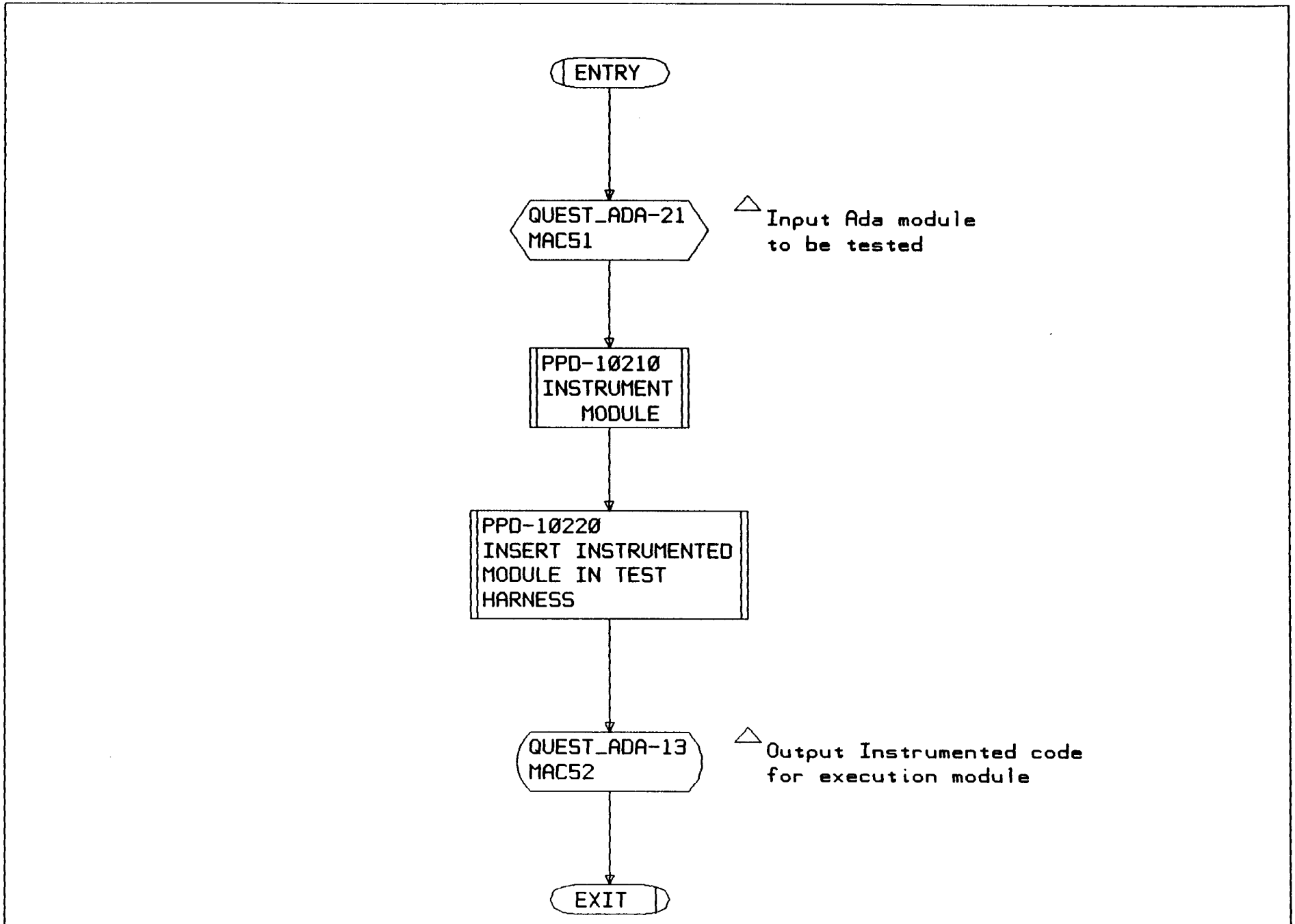
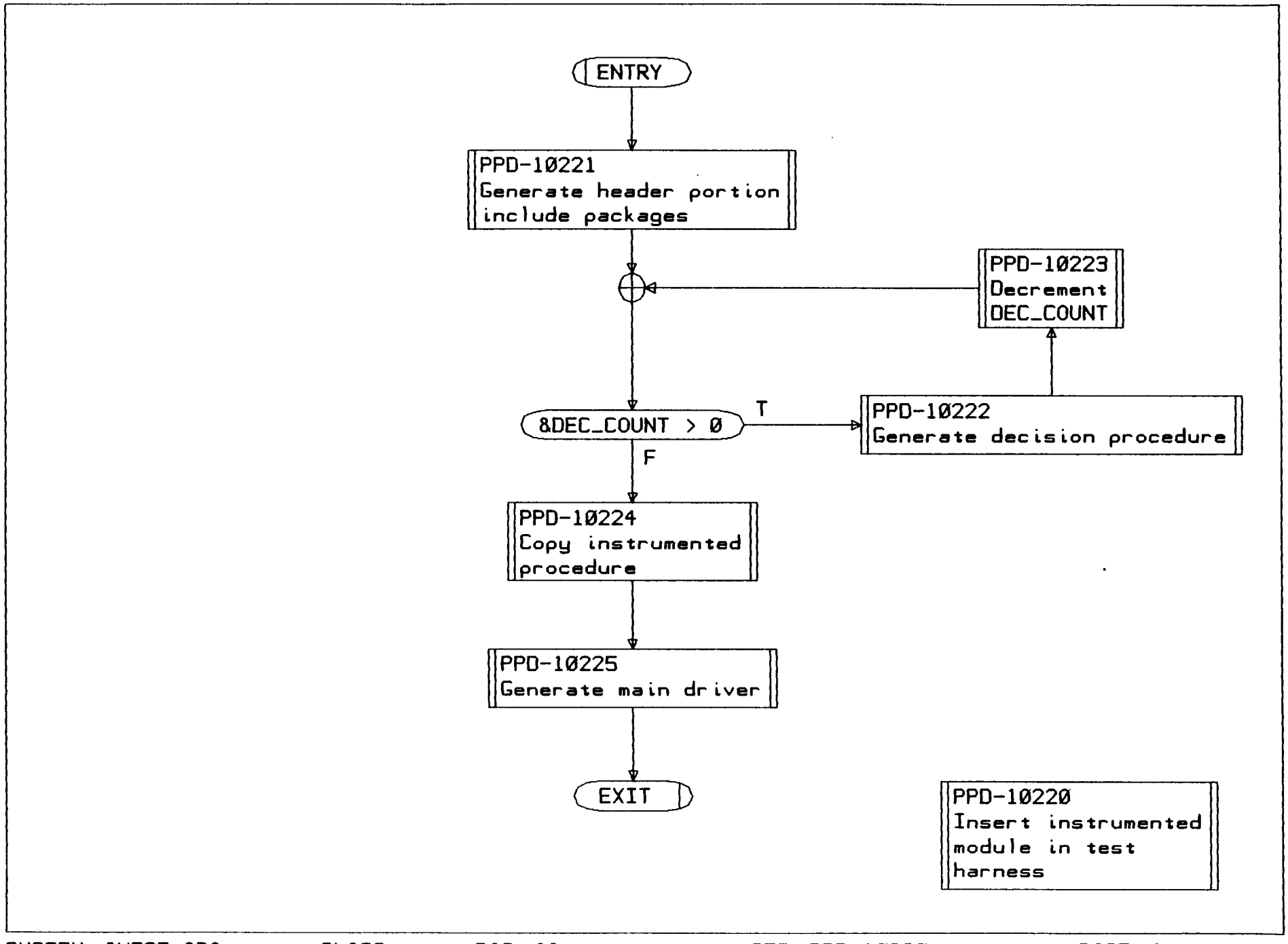


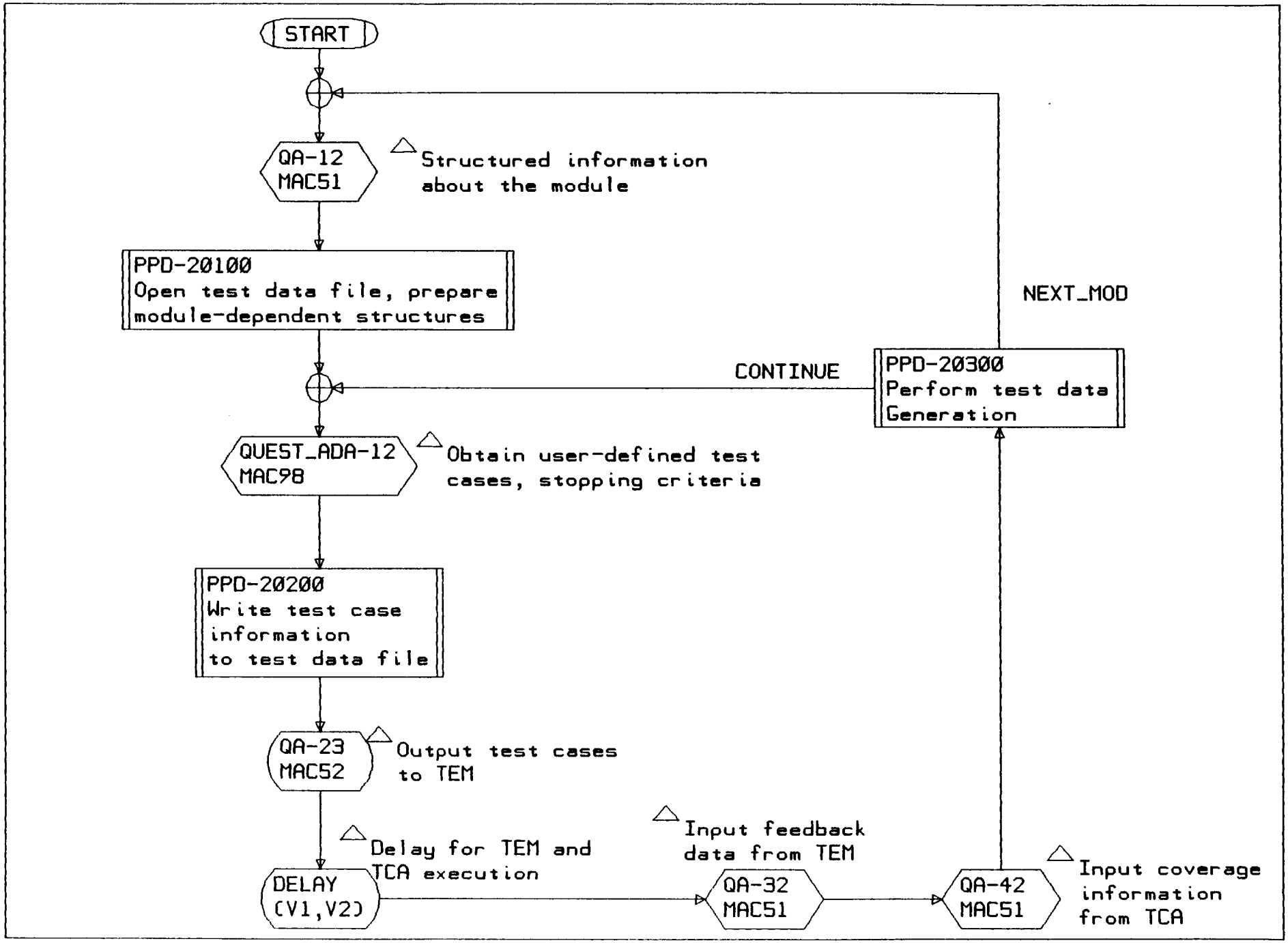
FIGURE 2. - SECOND LEVEL SCHEMATIC BLOCK DIAGRAM

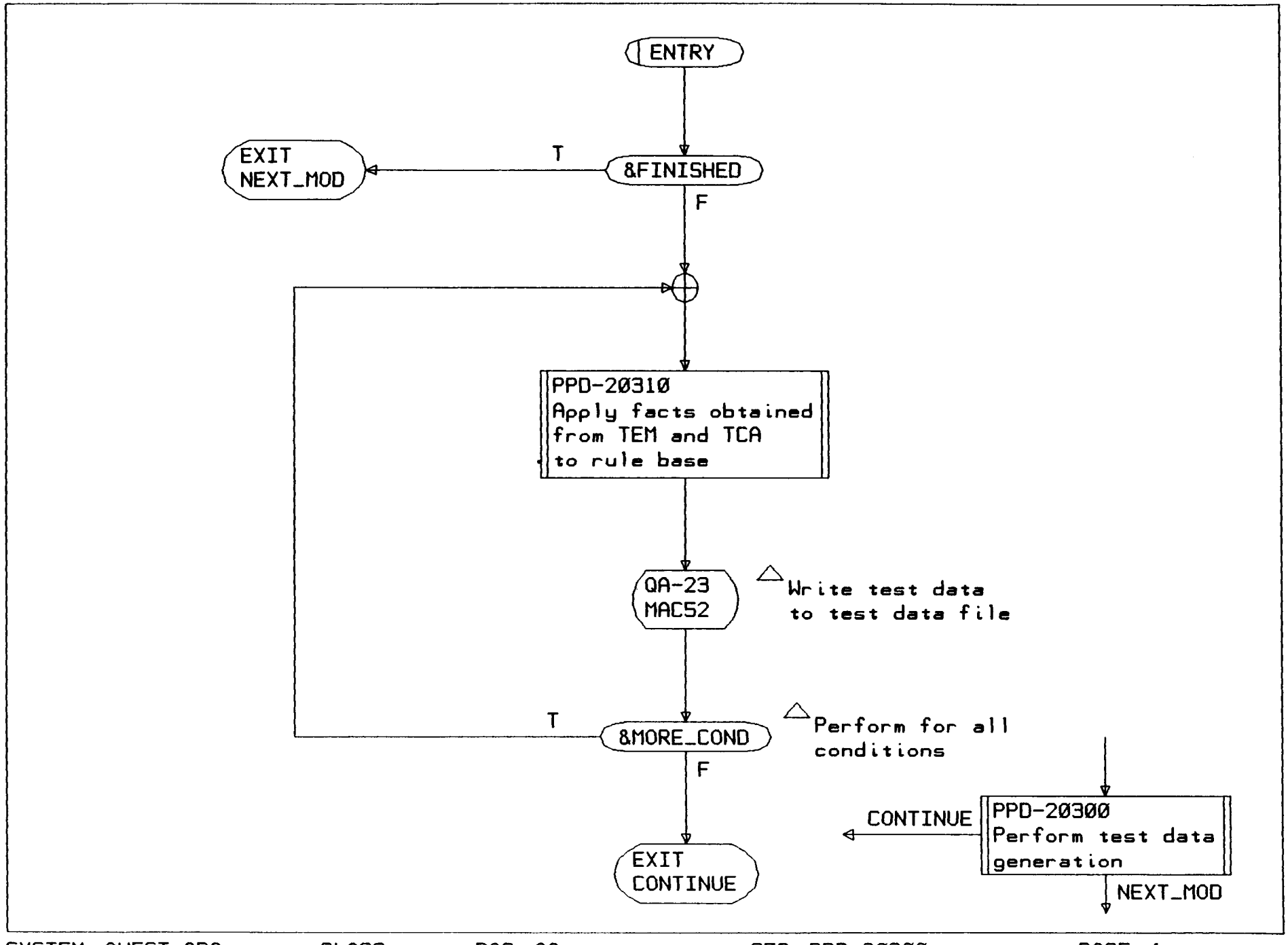

START
QUEST_ADA-12
MAC98
Input whether the parse/scan is for system or module
T
&NEWSYS
F
PPD-10100
Initial parse/scan of the total system under consideration
PPD-10200
Detailed parse/scan of a specific Ada module

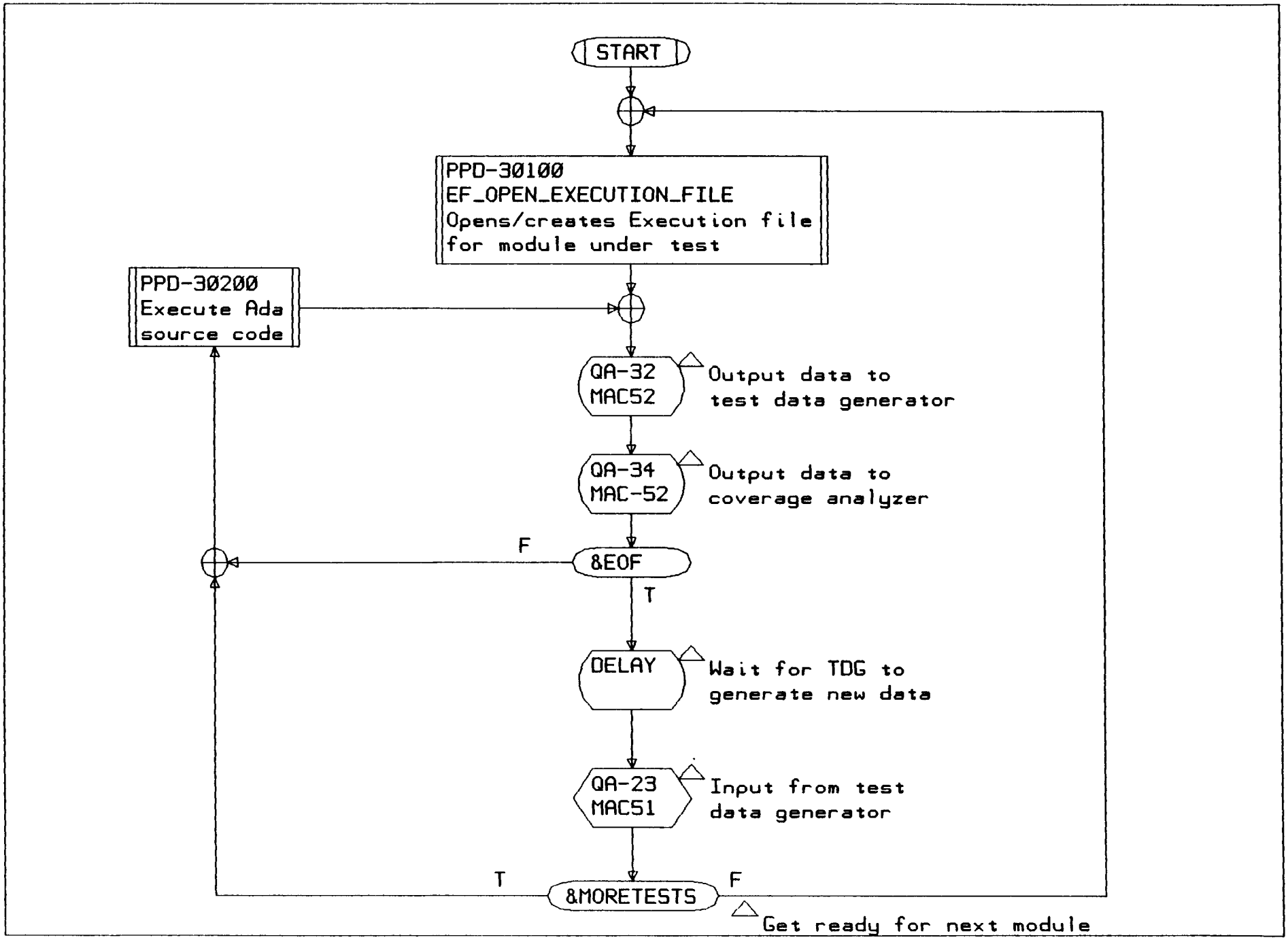

START
QUEST_ADA-12
MAC51
Input raw Ada code
for total system
PPD-10110
BUILD SYMBOL TABLE
QA-12
MAC52
Output initial information
for test data generator
QA-15
MAC52
Output initial information
for report generator
EXIT

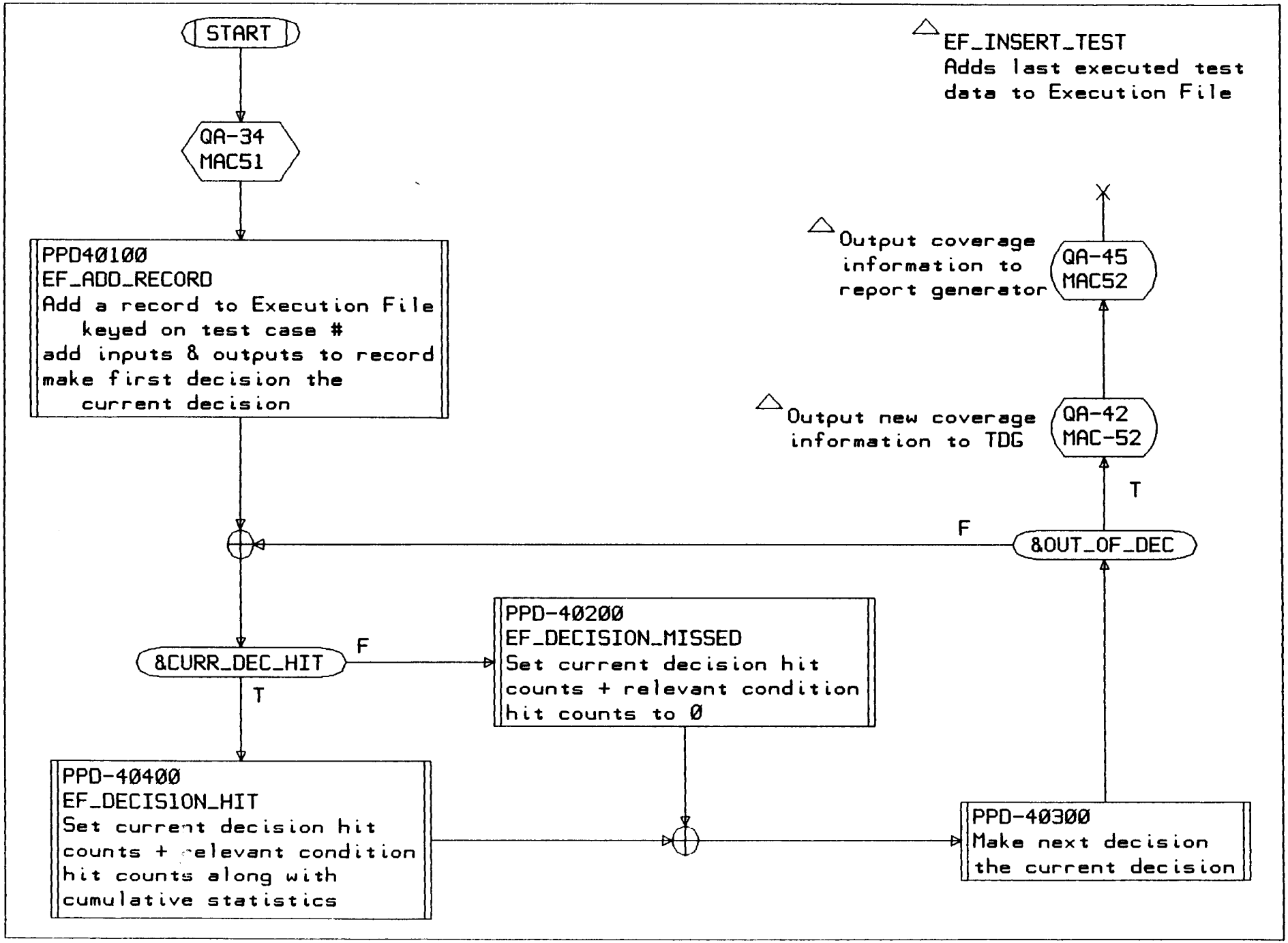

ENTRY
QUEST_ADA-21
MAC51
Input Ada module
to be tested
PPD-10210
INSTRUMENT
MODULE
PPD-10220
INSERT INSTRUMENTED
MODULE IN TEST
HARNESS
QUEST_ADA-13
MAC52
Output Instrumented code
for execution module
EXIT


ENTRY
PPD-10221
Generate header portion
include packages
PPD-10223
Decrement
DEC_COUNT
&DEC_COUNT > 0
T
PPD-10222
Generate decision procedure
F
PPD-10224
Copy instrumented
procedure
PPD-10225
Generate main driver
EXIT
PPD-10220
Insert instrumented
module in test
harness


START
QA-12
MAC51
Structured information about the module
PPD-20100
Open test data file, prepare module-dependent structures
NEXT_MOD
CONTINUE
PPD-20300
Perform test data Generation
QUEST_ADA-12
MAC98
Obtain user-defined test cases, stopping criteria
PPD-20200
Write test case information to test data file
QA-23
MAC52
Output test cases to TEM
Delay for TEM and TCA execution
Input feedback data from TEM
Input coverage information from TCA
DELAY
(V1,V2)
QA-32
MAC51
QA-42
MAC51


ENTRY
EXIT
NEXT_MOD
T
&FINISHED
F
PPD-20310
Apply facts obtained
from TEM and TCA
to rule base
QA-23
MAC52
Write test data
to test data file
T
&MORE_COND
Perform for all
conditions
F
EXIT
CONTINUE
CONTINUE
PPD-20300
Perform test data
generation
NEXT_MOD


START
PPD-30100
EF_OPEN_EXECUTION_FILE
Opens/creates Execution file
for module under test
PPD-30200
Execute Ada
source code
QA-32
MAC52
Output data to
test data generator
QA-34
MAC-52
Output data to
coverage analyzer
F
&EOF
T
DELAY
Wait for TDG to
generate new data
QA-23
MAC51
Input from test
data generator
T
&MORETESTS
F
Get ready for next module


START
QA-34
MAC51
PPD40100
EF_ADD_RECORD
Add a record to Execution File
keyed on test case #
add inputs & outputs to record
make first decision the
current decision
EF_INSERT_TEST
Adds last executed test
data to Execution File
Output coverage
information to
report generator
QA-45
MAC52
Output new coverage
information to TDG
QA-42
MAC-52
T
F
&OUT_OF_DEC
&CURR_DEC_HIT
F
T
PPD-40200
EF_DECISION_MISSED
Set current decision hit
counts + relevant condition
hit counts to 0
PPD-40400
EF_DECISION_HIT
Set current decision hit
counts + relevant condition
hit counts along with
cumulative statistics
PPD-40300
Make next decision
the current decision

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
Intergraph

David B. Brown
and
Kai-Hsiung Chang
Auburn University

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

November 15, 1988

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.

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. For this reason, continued research is needed in the area of software testing.

The property of perfect program correctness is difficult (if not impossible) to define and evaluate. 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 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. 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 very well understood [MIL84,

PAN78].

The approach which typically has been taken is to try 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 cannot yield acceptably 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 will not provide the necessary coverage of the program. This paper demonstrates that a heuristic rule-based approach to test data generation can easily produce a large amount of test data which will provide a much greater degree of coverage than randomly chosen data.

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". The first formal conference on software testing took place at the University of North Carolina in June 1972. Since that time, testing research has continued on several fronts, including the automation of portions of the testing process.

Software testing, as referenced in this document, is strictly dynamic testing, which is the execution of programs with specific input data and the production and assessment of outputs [WEY86]. This type of

software validation takes place in the programming and maintenance phases of the software life cycle. It is recognized that testing and validation techniques must be employed also during the requirements definition and design specification phases, as the cost of fixing bugs is higher the later they are uncovered in the software life cycle [HOW82]. 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, while an error is an incorrect output resulting from a fault.

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 [HOW86]. Some kind of 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 already mentioned are design specification simulators and redundant manual code implementations.

Some type of test adequacy criterion is needed to determine 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 how well 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 example, a simple instrumentation scheme could insert counters to record the number of times each statement is executed.

Test data generation has been defined as "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, random testing should not satisfy test adequacy criteria as well as would selectively chosen test data. On the other hand, algorithms for generating test data to satisfy particular adequacy criteria have generally had very bad time and space complexities, thus producing small amounts of test data. In fact, it is in general not possible (that is, there exists no algorithm) to generate test data which causes the execution of an arbitrary program path [MIL84]. This is the predicate solution problem, which reduces to the halting problem. 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 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 [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. 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 very appropriate in 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 very important to the rule base presented in this paper.

THE TEST DATA GENERATION PROBLEM

Test data generation algorithms are usually designed to generate test data sets which satisfy some particular test adequacy criterion, such as statement coverage. Since algorithms such as these are probably nonexistent for a general program, the domains of the algorithms are some subset of all possible programs, e.g., 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 one problem with conventional test data generation algorithms. A 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

Query Utility Environment for Software Testing (QUEST) [BRO86, WEY88a, WEY88b]. 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. 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. Before describing this model, it is necessary to have a firm criteria for developing the rule base. This is described in the next section.

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, 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 to be sure that they are functionally correct, the satisfaction of stronger criteria also provide more evidence of the correctness of the program under test. For these reasons, the strongest adequacy criteria were chosen to provide the best basis for rule development: path boundary domain coverage and multiple condition coverage. The other criteria are significantly weaker than these. Thus, the rules which were developed attempted to define a procedure by which the test cases generated would satisfy path boundary domain coverage and/or multiple condition coverage.

A test data generation rule 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.

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.

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 [DEM78] and Howden [HOW86], 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, $(v_1, v_2, \ldots, v_n)$. 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, say COND, involves only the ith variable. Its best test case $(v_1, v_2, \ldots, v_i, \ldots, v_n)$ 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 input parameters.

Rule 0 provides the starting values for test data generation. When the automatic test data generator is used to test code, these starting points will not be random; rather, they will 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.

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 [HOW86]. 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.

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 [HOW86].

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 very desirable when the opposite outcome has not been covered by any previously executed test case. This general approach was used quite successfully by Prather [PRA87].

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.

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 for evaluation purposes. The prototype is applicable to a subset of VAX Ada.* The Ada subset, as described in the following section, defines the scope of the prototype. Subsequent sections discuss the parser requirements, rule interpretation, test execution, and coverage evaluation portions of the prototype. The reason for implementing a prototype was to evaluate the ability of a rule-based test data generator to produce good test cases.

The scope of the prototype implementation was limited in two major

---

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

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 packageSTANDARD. 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 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. This is performed by a specialized parser/scanner developed for this purpose.

RULE INTERPRETER

The computer program which controls the entire prototype testing process was written in Prolog. At the highest level, it reads in the information about a program and repeatedly generates test cases and

calls a driver program to execute these test cases until it has done so the number of times chosen by the human operator of the program. Once the Prolog interpreter is activated, it consults a separate Prolog file which contains the test data generation rules. Then it queries the user for the name of the procedure to be tested, the number of iterations, and the maximum number of test cases to be generated during a single iteration. It then evaluates clauses to construct the names of the symbol file, the execution file, and the test case file for the procedure to be tested. The next step causes all applicable rules to fire. The test cases generated are placed in the test case file, and control is passed to the driver program of the procedure being tested. When control returns, the execution results file is consulted and the success of each test case executed is evaluated based on the execution results. The last action is to succeed (stop) if the desired iterations have been performed; otherwise this procedure recursively calls itself to continue the testing process.

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 "driver" program, that is, a program which reads the test file, executes the procedure, and records the results of the execution in a file. This process is repeated once for every test case in the test case file. The driver produces an execution file which is the feedback into the test data generator.

The driver consists of two parts: 1) the procedure being tested and 2) the instrumentation procedures, which measure coverage. The

driver algorithm is quite simple, and is (in pseudocode):

```
repeat for all tests in test case file
    initialize coverage matrix
    execute procedure under test with test case
    output 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. While the module driver and instrumentation could be generated by commercial Ada parser/scanners, currently this is not done, and their proprietary nature makes their augmentation impossible. For this reason a specialized parser/scanner is being constructed for this purpose. In addition to its producing the instrumentation/driving mechanism, the parser/scanner is also producing information to fire the rules in the rule-base, as described above.

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. Figure 1 shows a data-flow diagram of the rule-based test data generation system. Briefly, the rule interpreter reads the rule base

and symbol files, generates test cases, and writes these to the test case file. The module driver reads each test case, executes the module under test, and records the results in both results files. The Prolog-readable results file is used by the rule interpreter to generate more test cases, and the entire process continues for a user-selected number of iterations. At this point, the human-readable results file is examined to determine the coverage achieved. The coverage metrics computed are condition coverage, decision coverage, multiple-condition coverage, and three variants of each of these metrics concerned with domain boundary coverage.

Table 1 shows some statistics about the four Ada procedures 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 very complex and quite difficult for automatic solution using predicate solution techniques.

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 2 shows a comparison of the coverage achieved by the prototype test generator and a random test data generator. Each row of this table represents a single test suite. The first column of each row indicates the program under consideration. The size of each test suite is given in the second column. The remaining columns indicate the number of coverage obtained (e.g., 21 conditions covered out of 24 possible conditions = 87.5%).

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

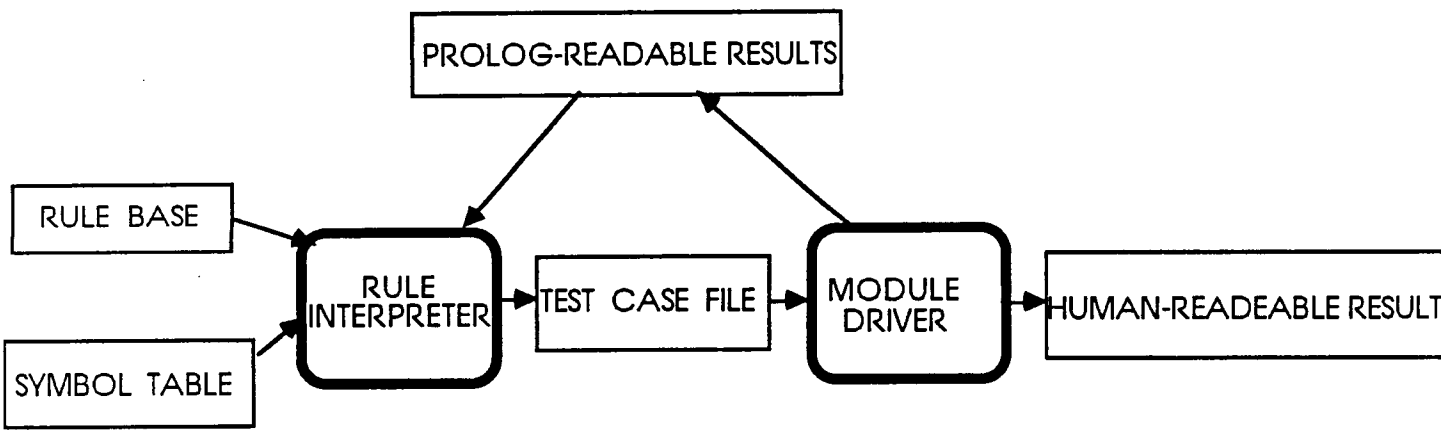


Figure 1. Rule Based TCG Paradigm

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

Program Method Used	Tests	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)

Program	Method Used	Tests	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

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.

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, is having the format of: <expl> <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      <expl> does not contain variables
then    exchange positions of <expl> and <exp2>
```

Rule B

```
If      <expl> contains constants
then    move all possible constants to <exp2>
```

These rules would simplify <expl> such that it contains at least

one variable and no constants. For example given a condition

```
3  =<  5 * X + 4

<expl>: 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 <expl> and <exp2>, after simple simplification steps the combinations can be generalized into the following 10 cases.

	<expl>	<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 case 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
```

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

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

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

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

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

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

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

```
test data: 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.

CONCLUSIONS

The main goal of this paper was to demonstrate the feasibility of a rule-based software test data generator. 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. Also, 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 it is expected to 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

[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.

[BRO86] Brown, D. B., "Test Case Generator for TIR Programs", Contract Number DAAH01-84-D-A030 Final Report|, September 30, 1986.

[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.

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

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

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

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

[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 Trans. Software Engineering|, Vol. SE-12, No. 10, October 1986.

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

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

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

[NTA88] Ntafos, S. C., "A Comparison of Some Structural Testing Strategies", IEEE Trans. Software Engineering|, Vol. 14, No. 6, June 1988.

[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 Trans. Software Engineering, Vol. SE-13, No. 7, July 1987.

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

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

[ROS85B] Ross, S. M., "Software Reliability: The Stopping Rule Problem", IEEE Trans. Software Engineering|, Vol. SE-11, No. 12, Dec. 1985.

[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.

[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, Apr 20-22, 1988, Mobile, Alabama.

[WEY86] Weyuker, E. J.,"Axiomatizing Software Test Data Adequacy", IEEE Trans. Software Engineering|, Vol. SE-12, No. 12, Dec. 1986.

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

FOOTNOTES

W. H. Deason is with Intergraph in Huntsville, Alabama.

D. B. Brown and K. H. Chang are with the Department of Computer Science and Engineering at Auburn University.

This work was partially supported 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

Fig. 1 Rule Based Test Case Generator Paradigm