

QUEST/Ada

QUERY UTILITY ENVIRONMENT FOR SOFTWARE TESTING OF ADA

*MURSHIEL
IN-61-CL*

**The Development of a
Program Analysis Environment
for Ada**

Contract Number NASA-NCC8-14

Task 1, Phase 2 Report

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

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

November 1989

TABLE OF CONTENTS

	<u>PAGE</u>
Acknowledgements	
1.0 Executive Summary	1
2.0 Literature Review: Concurrency Testing	4
2.1 Overview of the Literature Review	4
2.2 Significant Article List	4
2.2.1 Static Analysis	4
2.2.2 Task Monitoring	5
2.2.3 Testing/Debugging	5
2.2.4 Optimization of Analysis	6
2.3 Detailed Survey of the Literature	7
2.3.1 Introduction	7
2.3.2 Representation of Concurrency	7
2.3.3 Static Concurrency Analysis	11
2.3.4 Symbolic Execution	12
2.3.5 Task Monitoring/Interaction	13
2.3.6 Testing Concurrent Programs	20
2.3.7 Optimization of Analysis	20
2.3.8 Conclusion	21
3.0 Prototype Development	22
3.1 Overview of the QUEST/Ada Prototype	22
3.2 Test Data Generator	24
3.2.1 Best Test Cases	24
3.2.2 Test Data Generator Procedure	30
3.2.2.1 Fixed Percentage Modification	31
3.2.2.2 Random Modification	31
3.2.2.3 Condition Constants Modification	31
3.2.3 Test Case Generation Rule Organization	31
3.3 Parser/Scanner	33
3.3.1 Basic Instrumentation	33
3.3.2 Instrumentation for Symbolic Evaluation	34
3.3.3 Instrumentation for Multiple Conditions	34
3.3.4 Automatic Instrumentation	34
3.4 Coverage Analyzer	35
3.4.1 Autotest and the Test Coverage Analyzer	36
3.4.2 TCG for Compound Decisions	38
3.4.2.1 Test Case Goodness Measures	38
3.4.2.2 Test Case Generation Rules	39
3.5 Symbolic Evaluator	40
3.5.1 Boundary Computation	40
3.5.2 Facts Used by the System	42
3.5.3 Saliency Levels of Rules	44
3.5.4 Control Flow	44
3.5.5 An Example	45
3.5.6 Symbolic Evaluator Interface	46
3.6 Librarian	46

3.6.1	Basic Concepts	47
3.6.2	Using the Librarian	48
3.6.3	Details of the Code	49
3.6.4	BPLUS Portability Notes	50
4.0	Experimental Evaluation	51
4.1	Experimental Design	51
4.2	Experimental Results	52
5.0	Project Schedule	53
6.0	References	55

Appendix A --- Paper: Automated Unit-Level Testing with Heuristic Rules by Carlisle, Chang, Cross and Keleher

Appendix B --- Example of Instrumented Programs

Appendix C --- Librarian Routines

ACKNOWLEDGEMENTS

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

FACULTY INVESTIGATORS

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

GRADUATE RESEARCH ASSISTANTS

Kenneth Bradley
John R. Huggins
William R. A. Keleher, Technical Manager
Eric Richards
Benjamin B. Starke
Kevin Sullivan

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

1.0 EXECUTIVE SUMMARY

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

Phase 1 of this task completed the overall QUEST/Ada design, which was subdivided into three major components, namely: (1) the parser/scanner, (2) the test data generator, and (3) the test coverage analyzer. A formal grammar specification of Ada and a parser generator were used to build an Ada source code instrumenter. Rule-based techniques provided by the CLIPS expert system tool were used as a basis for the expert system. The prototype developed performs test data generation on the instrumented Ada program using a feedback loop between a test coverage analysis module and an expert system module. The expert system module generates new test cases based on information provided by the analysis module. Information on the design is given in the Phase 1 Report, dated June 1, 1989, and these details will not be repeated here.

The current prototype for condition coverage provides a platform that implements expert system interaction with program testing. The expert system can modify data in the instrumented source code in order to achieve coverage goals. Given this initial prototype, it is possible to evaluate the rule base in order to develop improved rules for test case generation. The goals of Phase 2 follow:

1. To continue to develop and improve the current user interface to support the other goals of this research effort (i.e., those related to improved testing efficiency and increased code reliability),
2. To develop and empirically evaluate a succession of alternative rule bases for the test case generator such that the expert system achieves coverage in a more efficient manner, and
3. To extend the concepts of the current test environment to address the issues of Ada concurrency.

The remainder of this summary will briefly describe the progress in accomplishing these goals according to the order given in the report.

A major literature review was conducted with regard to the testing of code which supports concurrency. This is given in Section 2 of the report organized according to the major issues within concurrency testing. Significant articles were found in the areas of: (1) static analysis, (2) task monitoring, (3) testing/debugging, and (4) improving the efficiency of the analyses (optimization). The literature review clearly revealed that static analysis is expensive to perform on complex tasking programs. However, if the amount of tasking used is simple and easily managed, static analysis can be used to provide an initial knowledge of the task state space.

A second major finding of the literature review was that a run-time monitor, possibly with task scheduling capabilities, should be integrated into the design of QUEST/Ada. Task monitoring is essential in studying concurrent tasks. This requires transformation of the original program into a new program that calls the task monitoring prior and after tasking activities. While this is analogous to instrumentation, the issue of test data generation is complicated by concurrency. In addition to path coverage, concern must be with concurrent history coverage, since the same input space could produce different outputs when executed through different concurrent histories.

The literature review also revealed that the main advantage of concurrency analysis is that it provides insight into the tasking interactions with concurrent programs. By using the monitor task and by examining the potential concurrent histories, many tasking logic errors can be identified. However, the major errors that the analysis purports to find, rendezvous deadlock and shared variable parallel update, would not occur in an Ada program that uses Ada's advanced tasking features that were especially designed to avoid these problems. As the design extension to accommodate concurrency evolves during the second half of Phase 2, strong consideration will be given to adopting a practical view of concurrency as it is currently being applied to NASA applications.

The prototype developed in Phase 1 has continued to evolve in order to collect data to determine the viability and effectiveness of the rule-based testing paradigm. This prototype consists of five parts, which are discussed in Section 3 of this report. Special emphasis has been given to the Test Data Generator (TDG), the expert system designed to select the test data that will be most likely to drive a specific control path in the program. Four types of rules have been used in the development of the TDG: random, initial, parse-level, and symbolic evaluation. Random rules provide base data for the more sophisticated rule types to manipulate. Initial rules generate simple base data from the

information supplied from the parse. Parse-level rules, which are more sophisticated, rely upon the coverage table and best-test-case list developed by the Test Coverage Analyzer. Symbolic evaluation rules extend this concept by representing each section of the program as an abstract function. The symbolic evaluation rules utilize the coverage table and the symbolic boundary information provided by a symbolic evaluator.

The more sophisticated rule types rely on the Test Coverage Analyzer (TCA), which has had to undergo corresponding modification. The TCA provides two major functions: maintaining the coverage table, and determining the best test case for every decision. This information is used by the parse-level and symbolic evaluation rules to determine which decisions or conditions need to be covered to provide complete decision/condition coverage. The best test case for each decision is determined by a mathematical formula describing the closeness of a given test case to the boundary of a specific condition. The test data generator rule bases modify the best test case to attempt to create new coverage in the module under test.

Work has also been initiated on a Symbolic Evaluator (SE), which uses detailed information about the source code being tested to attempt to represent each path through the code as an abstract function. The work of the symbolic evaluator is divided into two parts -- developing and evaluating symbolic expressions. Using descriptions of the conditions in the module under test provided, the SE develops symbolic boundary expressions in which each of the variables in a condition is represented in terms of the other variables. After developing the symbolic boundary equations, the SE evaluates them using the test data as it appears at the time the condition is executed.

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

Section 4 of this report presents the experimental design for the evaluations which are anticipated in the second half of this phase. These results will reveal the validity of the rule-based approach toward test case generation in that a comparison between each successive set of rules will be performed as they evolve. Since the prototypes have been brought to a state where tests can be run in the near future, these results are expected within the next few months. Section 5 presents a review of the project schedule and the anticipated results from this phase.

2.0 LITERATURE REVIEW: CONCURRENCY TESTING

2.1 OVERVIEW OF THE LITERATURE REVIEW

This chapter of the report is a working draft of the concurrency testing literature review. It makes frequent reference to the bibliography of collected papers, which is contained in Section 2.4. The first subsection is a brief summary of significant articles, which begins with static analysis, moves on to dynamic task monitoring, covers other testing/debugging topics, and then ends with notes on optimization of the analysis. A second subsection goes into considerable detail on these respective topics.

2.2 SIGNIFICANT ARTICLE LIST

2.2.1 STATIC ANALYSIS

Generally, static analysis leaves much to be desired. It has some highly restrictive rules stemming from its inability to deal with dynamic tasks or subscripted references to tasks. Also, it considers too large a sample space (this is especially true of Taylor's work). The analysis of large amounts of tasking information consumes a huge computational overhead. Static analysis is usually best for finding relatively simple mistakes which probably would not occur in code created by professionals who use Ada's advanced tasking features. Significant articles on static analysis related to tasking include:

- [Tayl80] This is a precursor to [Tayl83b], in which errors are detected in a program via data flow analysis. The language considered is a derivative of HAL/S. Taylor later rebukes this paper for (1) not using Ada as the target language, and (2) not having sufficient generality.
- [Tayl83b] This article presents an algorithm for analyzing concurrent tasks. While the algorithm has its faults, it is recognized as the standard for an introductory approach to static analysis of concurrent programs.
- [Tayl88] This is the sequel to [Tayl83b]. In this paper, Taylor presents methods to: (1) make the sample space considered by his algorithm more "correct" via symbolic execution, and (2) optimize the selection of the sample space for the algorithm.

- [Call89] This paper focuses on the creation of a data flow framework based on the analysis of programs for the following constructs: synchronization, sequential execution, data dependence, and execution order.
- [Stran81] An approach is presented which uses language theory to help create a static notation for inter-process communication for keeping track of tasking activity.
- [Mura89] "Petri net invariants" are employed to detect Ada deadlocks statically.

2.2.2 TASK MONITORING

The field of task monitoring has developed into a useful tool. This approach requires that the source program be transformed into a new program with embedded calls to a run-time monitoring task. This monitor can detect deadlock before it occurs and can provide a tasking event history to trace what occurred to cause an error in the program. Tracing is also used to note the history of "correct" execution. The biggest concern with monitoring is making sure that the modified program is computationally equivalent to the original source program and that the translation does not conceal potential errors. A number of task monitors have been implemented for Ada. Suggested references include:

- [Helm85] An Ada tasking monitor implementation is presented.
- [Chen87] The EDEN execution monitor for Ada tasking programs is reviewed.
- [Gait86] This paper reviews the probe effect, i.e., the insertion of time delay calls into the code. If variation of duration of these time delay probes cause the program to act differently or to produce different results, then it can be reasoned that in all likelihood the program is highly dependent on the timing of its execution. Note that the introduction of probes can also be used to "force" crude scheduling.
- [Germ84] The three main topics considered by this paper are: correctness of program transformation into a monitored program, the duties of the monitor task, and a method for producing unique task identifiers.

2.2.3 TESTING/DEBUGGING

The testing of concurrent programs involves much more than just providing a test set of data. Due to the nature of programs executing concurrently (or in parallel), different results may be

produced for the same test set of data. Therefore, in addition to testing the concurrent programs, there must be a way to establish the execution sequence if the programs are to be tested effectively. Before this synchronization of execution can be developed, however, the underlying concurrent structure of the tasks has to be understood.

It is imperative that testers be able to study the results of a test set. Monitoring, therefore, is a precondition to concurrent program testing, since the output of the task monitor allows post analysis of the test data performance. The following references apply:

- [Tai85] The problem addressed is that of a concurrent program producing different results when executed multiple times with the exact same input. The concept of an IN_SYN test case to establish synchronization is presented.
- [Gold89] This paper establishes that concurrency activity can be divided into language specific and language independent categories. Information gathered by a run time monitor can be studied off-line to gain insight into the behavior of the concurrent programs.
- [Hseu89] The concept of concurrent data path expressions is presented. Their goal is to aid in the revelation of the underlying concurrent interrelations in a set of tasks.
- [Brin89] The main focus of this paper is the development of a debugger for testing Ada tasking programs. It makes several interesting points in stating the requirements for transforming a program into a state that allows: (1) control over the sequence of execution of the program and, (2) investigation into the current status of the program during execution.
- [Ston89] The concurrency map representation is created to aid in the understanding of the interrelations between concurrent tasks.

2.2.4 OPTIMIZATION OF ANALYSIS

The implementation of Taylor's simple static analysis algorithm for concurrent tasks has the unfortunate property of combinatorial explosion. The analysis theory itself, however, can be augmented with a number of optimization rules to limit the amount of space that has to be considered by the analyzer, thus reducing the amount of output. Optimization also strives to prune out generated concurrency states that, although theoretically possible, cannot occur due to the logic of the program.

Taylor [Tayl83b, Tayl88] presents both the static analyzer and the methods to improve the analyzer. The methods include: (1) reducing the amount of tasks considered at a given moment (parceling), and (2) employing "run-time" scheduling to decide which states are not possible.

2.3 DETAILED SURVEY OF THE LITERATURE

2.3.1 INTRODUCTION

When confronting the literature with regard to testing code involving concurrency, new issues arise from those which are established in classical testing theory. In addition to detecting faults common to non-concurrent code, three major goals emerge: (1) find possible deadlocking, (2) find possible shared variable parallel access/update, and (3) test the program through different concurrent states. The following subsections address the literature on concurrency testing by organizing and summarizing it into six classifications. First, the **representation of concurrency** is considered in terms of the different modeling schemes employed. Then **static concurrency analysis** will be discussed in terms of its advantages and drawbacks as well as the different modeling schemes which exist.

Symbolic execution will be considered next, and the reasons for and results of its use will be given along with the overall scheme employed. At that point, **task monitoring/interaction** will be defined in terms of its capabilities and the problems involved in using it. This will set the stage for the major works on **testing concurrent programs**, in which the following issues will be considered: (1) How is data generated for symbolic programs? (2) What should be saved from one generation of results from a test? (3) Given previous test cases, what should be saved to modify the next test case? and (4) What static analysis results are required for dynamic analysis? In a final subsection conclusions from the literature review will be summarized as they apply to the current project.

2.3.2 REPRESENTATION OF CONCURRENCY

Taylor's [Tayl83b] method produces all possible task state transitions for a number of active tasks. First, Taylor requires a specialized version of the program state graph whose nodes are related only to tasking (states that involve no tasking are coalesced). Taylor's algorithm then proceeds from the main task's beginning and puts all possible next states onto a stack. A state is popped off of the stack, and all the possible next states for it (if any) are put on the stack. The algorithm proceeds until

the stack is empty. Note that a record of the duplicate states is maintained so that infinite state loops are avoided.

Taylor defines the following as significant task events: (1) Entry call, (2) Accept statement, (3) Delay statement, (4) Abort statement, (5) Task declaration, (6) Declaration of data type/-object containing a task, and (7) Operation on objects shared by tasks. To generate the possible task states a program can execute, the following are used as the basis:

Program Call Graph: subprogram invocation structure, which indicates the subroutines each unit can call and the subroutines which can call each unit; and

Program Scope Information: nesting (hierarchical structure) of the program's constituents.

The following definitions are useful in understanding Taylor's work:

S	The program under test.
UNIT	Made up of elements, i.e., procedures, functions, tasks, and blocks contained in S.
U	The number of elements in the UNIT, $ UNIT $.
Call Graph(S)	The call graph of program S (CG(S)) consists of nodes P and directed arcs I that represent the potential for invocation within the program S. There is a direct relationship between the p_i nodes of P and the elements of UNIT. The arc (p_i, p_j) exists within I iff the unit that p_i corresponds to can invoke the unit p_j represents. Invocation may occur if: (1) p_j is a subprogram that p_i may call, or (2) p_j is a block inside p_i 's body.
TASKS	The set of all tasks t_i comprises program S. TASKS is a subset of UNITS. The main program is counted as a task.
T	The number in TASKS, $ TASKS $.
T'	The number of distinct tasks in S; T' might be greater than T for a program that has tasks declared in re-entrant/recursive subprograms.
Flowgraph	This is the directed flowgraph representation of S.
$\{G_1, \dots, G_U\}$	The set of flowgraphs for S, where $U = UNIT $.

G_i	Defined as (N_i, A_i, r_i) , this represents the flowgraph for a given individual unit within program S.
N_i	The set of nodes in G_i (represents a tasking event).
A_i	The set of arcs in G_i (representing flow of control from N_i).
r_i in N_i	The root node for the particular unit's flow.

Given these building blocks, the remainder of the analysis is concerned with finding the successor nodes for each state node in the flowgraph (a state node is one which performs a tasking-related activity). The set of successors for a given node are essentially those nodes in the flowgraph from which an arc emanates to the given node. The following definitions help in understanding the concurrency and successor concurrency states:

C	A concurrency state, which is an ordered T' tuple $(c_1, c_2, \dots, c_{T'})$, where each c_i is either a state node of a flowgraph G_j or is inactive. This can be considered a snapshot of the states of all possible tasks in program S.
C'	A successor concurrency state to C (there can easily be more than one). This can be successor if: <ol style="list-style-type: none"> (1) For all i, $1 \leq i \leq T'$, either: <ol style="list-style-type: none"> (a) c'_i in $\text{succ}(c_i)$, (b) $c'_i = c_i$, (c) $c_i = \text{inactive}$ and $c'_i = \text{begin task}$, or (d) $c_i = \text{end task}$ and $c'_i = \text{inactive}$, and (2) There exists at least one c'_j, $1 \leq j \leq T'$, which represents application of case (a), (b), or (d) above (thus requiring forward movement).

Given the definition for a concurrency state and the method for determining valid successor states, all possible concurrency states can be found for program S. Note that an individual instance of tasks states through a single execution of program S is called a concurrency history. This is further defined by the following:

$CH(S)$	Concurrency history of program S that is a sequence C_1, C_2, \dots, C_k of concurrency states such that: <ol style="list-style-type: none"> (1) $C_1 = (\text{begin } \langle \text{MAIN} \rangle, \text{inactive}, \dots, \text{inactive})$, and (2) For all i, $1 \leq i \leq k$, C_{i+1} in $\text{succ}(C_i)$.
---------	--

PH(S) Proper concurrency history for program S; an instance of a concurrency history for program S with the following restrictions: (1) the length of the history, k, is finite, (2) all of the states of the history are unique, and (3) the history is devoid of loops.

H(S) A set of all possible PH(S). This is the goal of the analysis: a collection of all possible progressions through the task states. Note that this represents distinct multiple executions of the program S.

Once H(S) has been generated, the concurrency states can be used for static analysis.

While Taylor's work must be considered the standard with regard to concurrency representation, Stone [Ston88, Ston89] has contributed the concept of time-line diagrams, where each task is represented as a line, and points on the line are tasking events. The lines are set up in parallel to one another and dependencies between tasks are shown by a directed arrow from one task's point to another task's point.

Stone also presented the concept of a concurrency map. Some task's events are unrelated, and the timing of their execution is unimportant. Some parts of a concurrent program, however, are time dependent and are known as interprocess interaction. According to Stone:

"The concurrency map expresses potential concurrency, and is both a data structure for controlling replay and graphic method of representing concurrent processes. The map displays the process histories as event streams on a time grid. Each column of the grid displays the sequential event-stream of a single process. The row represents an interval of time, and the events that appear in different columns in that row can occur concurrently." [Ston89]

Thus, the concurrent program is represented on two axis within the concurrency map. One axis (columns) represents a thread in the program, while the other axis (rows) represents forward movement through time. The event-stream (a program task) is made up of dependence blocks. Dependence blocks may have predecessor states which must occur in other tasks before the block may execute, and they may end with a successor state, signalling other event streams so that they can proceed. Also, normal non-concurrent related code can occur before and after a dependence block.

The rows that make up the concurrency map (associated with time) consist of concurrent events which all must complete before

the next row can be entered. A block may extend over more than one row. Time dependencies are shown in the map by an arrow starting at the end of one block and pointing to the beginning of another. The event-stream blocks may "float" up and down through time, as long as the extent of the movement through time does not go before or after any time dependencies associated with the block. This floating is known as map transformation. Three useful properties associated with map transformation are:

- (1) The collection of transformations of a map shows all the multiprocess event orderings that are consistent with the given time dependencies;
- (2) If two events in different processes are potentially concurrent, then there is a transformation of the map in which the two events appear in the same row; and
- (3) The map constructed from the process histories and the known dependencies is adequate in the sense that it represents all possibilities for concurrency.

Finally, in section three of the paper, it is demonstrated how the concurrency map could be used to represent a message passing concurrent system such as is the case with Ada. Also, Francesco [Fran88] presented a rather complex algebraic description of a tool for specifying and prototyping concurrent programs.

2.3.3 STATIC CONCURRENCY ANALYSIS

Taylor's work on concurrency representation extended into the static analysis of concurrent programs [Tayl83b]. Given the output of the task state generator algorithm, concurrent tasks can be analyzed for either deadlock or for parallel update of shared variables. Goals set for the static analysis include accuracy, minimization of superfluous error reports, and efficiency. Taylor notes the short-comings of static analysis:

- (1) inability to deal with referencing tasks by subscripting or pointers,
- (2) DELAY statements cause timing problems that cannot be resolved statically, and
- (3) dynamic task creation can cause an infinite number of ways to interpret program execution.

The following is summary of static concurrency analysis [TAYL88]:

"Static concurrency analysis builds a rooted directed graph of concurrency states. A concurrency state summarizes the control state of each of the concurrent tasks at some point in an execution, including synchronization information, while omitting other information such as data values. Directed edges in the concurrency state graph indicate which states may follow each other in executions of a program. A path from the

root node to any node in the graph is called a concurrency history since it captures a sequence of synchronization events that may occur in a program execution."

Taylor defined various sets containing concurrent action inter-relations. A concurrency history is one instance of a state transition through the program. If the history ends with tasks still active (perhaps waiting in an ACCEPT), then a deadlock state has been found. Individual states can be examined to see if two tasks can access/update a shared variable at the same time.

Taylor [Tayl88] stated that the main weakness of static analysis is that it can result in erroneous states -- task states that could not happen due to the logic of the program. These superfluous states can generate error messages for events that will not occur at run time. He presented a method in which static analysis and symbolic execution could be teamed together, which will be described in more detail below. In an earlier paper [Tayl83a] Taylor showed that analysis of concurrent programs is NP-hard. He also did some more general work on static anomaly detection [Tayl80].

Murata [Mura89] used Petri Nets as a static analysis tool to detect deadlocks in Ada programs. Callahan [Call89] presented some results involving the static analysis of low-level synchronization. Stranstrup [Stran81] and many others (see his references) performed some analyses of concurrent algorithms; however, the relationship between this work and that of testing concurrent programs is questionable.

2.3.4 SYMBOLIC EXECUTION

As introduced above, Taylor [Tayl88] has noted that static analysis alone is prone to errors. It generates all possible task state transitions, and therefore might generate task states that could not occur given the logic of the program. Therefore, he suggest an interaction between symbolic execution and static analysis, allowing one of them to work on the program for a while and then having the other take over. Symbolic execution serves to prune the information static analysis has generated. The two techniques can be combined in two ways: serial and interleaved.

In the serial application, static concurrency analysis is run first. After completion, all nodes that imply an error condition (deadlock or parallel variable update) are marked as "interesting." All of the ancestors of the interesting nodes are marked as "promising." Symbolic execution then produces its own graph. Promising states not existing in the symbolic execution graph are thrown out. Matching interesting states are marked as feasible. The process continues until either (1) all interesting nodes are marked feasible, (2) no more advancement can be made down a

promising path, or (3) some resource (e.g., CPU time) has been exhausted. The output of static concurrency analysis has now been pruned of incorrect error states [Tayl88].

When the two techniques are interleaved, one advances until it times out or until it requires the analysis of the other to advance. Static analysis begins and continues until either a possible error state is encountered or until some maximum number of nodes have been generated. Nodes on the "frontier" of static analysis are noted as being interesting, their ancestors being promising. Symbolic execution then takes over. Analysis is performed down only promising paths, with each node encountered under symbolic execution being changed from promising to feasible. When a node is reached with no children, analysis is suspended for later (in the event this node will indeed develop promising children). When static analysis resumes, it only processes those paths marked as feasible and promising.

Static concurrency analysis can be used to detect infinite waits as well as simultaneous updates of shared variables. By intertwining static analysis with symbolic execution, impossible conditions that would otherwise cause error messages can be avoided. Taylor recognized the weaknesses of using regular static analysis in dealing with dynamic objects, arrays indexed by expressions, and pointers. For static concurrency analysis, the following are problems:

1. Arrays of tasks;
2. Arrays of records that contain a task type as a member;
3. Pointers to tasks;
4. Recursiveness involving tasks.

With respect to complexity, Taylor has written a paper showing that static concurrency analysis is NP-hard. Given the basis of Taylor's work, the following approach is inferred in attempting to test concurrent programs:

1. Find a representation for tasking activity in the program,
2. Inter-relate static analysis with symbolic execution to remove impossible error states that are prone to manifest themselves in the concurrent static representation, and
3. Attempt to adaptively reduce the amount of space to be studied.

2.3.5 TASK MONITORING/INTERACTION

A major weakness of static analysis is that it requires consideration of all possible tasking states. This involves a huge amount of information to generate and to analyze. It also has certain restrictive rules which it applies to the sample space it

can consider (e.g., no dynamic tasks). A task monitor is a run-time supervisor that keeps track of the concurrency related states of the various tasks. By constantly analyzing the states of the task, it can detect when deadlock has occurred (or will occur), or when a variable can be accessed/updated in parallel. A monitor requires a preprocessor on the source code to insert calls to the monitor task.

The use of a monitor task is not without its problems. It may not provide an absolutely correct representation of the current tasking states. Introducing a monitor results in an overhead that may modify the program in such a way that certain errors will not be detected (a problem that does not exist in nonconcurrent testing). Also, there is difficulty in finding an easy representation for identifying a task for reports presented to the user.

Another use of a task monitor is to simulate discrete scheduling. Given static analysis output of all the possible tasking state transitions, this monitor could try to delay individual tasks in such a way that they progress according to a given concurrency history. Taylor [Tayl88] stated that a run-time supervisor is needed to make sure all possible task states are traversed. The run-time supervisor would be used to attempt to invoke specific task state procession. It could then monitor the various states of the tasks so that deadlock and parallel variable update/access faults could be detected.

Helmhold [Helm85] stated that a run-time monitor can detect a larger set of tasking errors than could static analysis. For Ada, he gave eight different task states:

- (1) Running,
- (2) Calling [enqueued, in rendezvous, circularly deadlocked],
- (3) Accepting,
- (4) Select-terminate,
- (5) Select dependents completed,
- (6) Block waiting,
- (7) Completed, and
- (8) Terminated.

In addition to each task's state, a list of its dependents is maintained.

A dead task is defined as a task that is blocked such that there is no possible way that it can become unblocked. A tasking state of a program is defined to be the set of tasks that have been activated by the program, their statuses, and any associated task information. A deadness error occurs in a program when its tasking state contains a dead task. Different deadness errors are:

- (1) Global blocking,
- (2) Circular deadlock, and

(3) Local blocking.

Helmhold stated that "determining if a program contains any deadness error is as difficult as the Turing machine halting problem."

A program must be modified in order to communicate with the monitor task. For identification of the tasks, an integer ID and a string identifier are created. The monitor creates a "picture" of the program's tasking state based on inserting entry calls to the monitor task at the following points: (1) before an existing entry call, (2) at the execution of an accept or select statement, (3) at the start or end of a rendezvous, (4) at the departure from a block, and (5) at the activation of a sub-task. Although this picture is updated whenever the monitor task is called, it is still possible that it will incorrectly represent the true tasking state of the program.

Whenever global blocking occurs, a snapshot of the program's tasking picture can be produced. The output includes the task string name, the task ID number, the status of the task, entry queue status, and task being called (if any).

After a lengthy demonstration of the use of the task monitor, Helmhold goes over possible extensions to this method. One calls for keeping track of more information (perhaps even entry call parameter values). In this implementation, it is known that a task has issued an entry call, but it is not known where the entry call was made (in relation to the source code). Keeping track of a complete state history for each task would allow "playback" to help decide where things started going wrong. Another extension is asking the user to play Oracle by specifying rules in tasking interaction (i.e., "This can never happen," or "This can only happen after this has happened..."). If one of the rules is broken, a user specified error has occurred.

Helmhold is to be credited as one of the few who have actually implemented a monitor and preprocessor (albeit without the mentioned extensions). As he stated [Helm85b], this monitor implementation suffers from some deficiencies. It does not work well with aborted tasks, prioritized tasks, or tasking statements executed during task elaboration. Deadness errors due to something other than rendezvous are not detected (e.g., shared variable communication).

Note that a monitor can be made to take evasive action since it can detect when deadlock is about to occur. Given this foresight, the monitor could raise an exception for deadlock.

It would be nice for the monitor to be part of the run-time scheduler. Then, the preprocessor would not be needed and data structures could be shared. However, by being separate, it allows

the monitor to (1) be independent of the scheduler's algorithm, (2) be portable since it is not associated with a specific implementation, and (3) team up the compile-time checker and the run-time monitor to look for deadness errors.

In converting the program P to the monitored program P', the following assumptions are made:

- (1) Every declarative region in P corresponds to a declarative region in P'.
- (2) Every declaration in P of a type of program unit (in the Ada sense) corresponds to a declaration in P' of the same kind.
- (3) Every object in P corresponds to an object or component object in P' of the same kind.
- (4) Every statement in P corresponds to a statement P' of the same kind.
- (5) Declarations, objects, and statements in a region R in P correspond to declarations, objects, and statements in the corresponding region R' in P'.

Program P and P' also have corresponding executions and equivalent potential errors. If the monitoring of P' is correct, then: (1) any possible deadness error in P also exists in P', (2) if deadness is detected, it happens before the error occurs, and the error will occur if the computation occurs normally, and (3) certain kinds of deadness errors will always be detected.

Although the monitor's picture of the tasking state of the program may differ from the actual state (whether due to early tasking notification or late tasking notification), a proof is presented to show that correct detection of error conditions occurs despite the differences. The article ends with an example of a monitor being performed on the dining philosopher's problem (the resulting transformed program appears in [Germ82]).

Cheng [Chen87] gives a presentation of EDEN, an event driven monitor for Ada tasking programs. To reduce the amount of interference the monitor task has on the tasking programs, EDEN employs the concept of "partial order preservation," which is based on lattice theory. EDEN provides tasking state snapshots and histories, interruption of program execution, and deadlock detection. It facilitates its processing by writing task histories to files.

To interact with the monitor, a given program P is transformed into program P'. Cheng asks the following three questions about monitoring execution: (1) What can be monitored at the Ada source code level? (2) How can information be collected about tasking behavior of the monitored program? (3) How can interference be reduced by the monitoring actions in order to guarantee the accuracy of the information reported by the monitor? Cheng lists the twenty-one possible states a task can be in:

- | | |
|-------------------------------|--------------------------------|
| (1) Starting Activation | (12) Block Completed |
| (2) Activating | (13) Block Termination Waiting |
| (3) Activated | (14) Block Terminated |
| (4) Executing | (15) Abnormal |
| (5) Delay | (16) Completed |
| (6) Entry calling | (17) Termination Waiting |
| (7) Accepting | (18) Terminated |
| (8) Selective Waiting | (19) Rendezvous |
| (9) Starting Block Activation | (20) Suspended by Rendezvous |
| (10) Block Activating | (21) Continue |
| (11) Block Activated | |

He states that "The life cycle of a task can be described by a sequence of states of the task from 1-Starting activation to 18-Terminated in terms of tasking behavior." Cheng criticizes [Helm85] for having so few tasking states since he feels that this does not present a complete picture.

A simple example of code transformation is shown for an ACCEPT statement. First, the monitor is called right before the ACCEPT to note that the task is "acceptable." After the ACCEPT has been engaged, another call is made to note that rendezvous is occurring. The statements of the accept entry are then executed. Right before the END for the ACCEPT, another call is made to note that the task is "continuing" and that rendezvous is at an end.

Cheng briefly notes that the "partial order preservation" concept keeps track of the way tasks proceed. He attempts to associate the transformed program back to the original (thus eliminating the effects of the monitoring task). He states that "We regard the program transformation as a mapping from the lattice for the original program to the lattice for the transformed program. If the transformation is homomorphic, then the partial order is preserved."

The EDEN implementation consists of a preprocessor (3000 source code lines) and a task monitor (6000 source code lines). The preprocessor keeps a symbol table of task type/objects so that it can realize when tasking interaction is occurring. The task monitor is in five parts:

- (1) Tasking-dynamic-dependence-tree: used to keep track of what frames (subprograms, blocks, or other tasks) a task depends on. Upon termination, the node is removed from the tree.
- (2) Entry-call-queue-manager: every time an entry call is made on a task, it inserts into a list which indicates who called the task and the time it was called. When rendezvous is complete or the call is aborted, the item is removed from the list.

- (3) Tasking-information-collector: this is a task whose entries correspond to all the different twenty-one task states. Each call is saved for later analysis.
- (4) Tasking-information-manager: saves information collected by the tasking-information-collector. It has exclusive read/write access to the information.
- (5) Query-processor: user interface that interprets commands.

In trying to find a unique identifier for each task, the DoD recommendation of using access values is rejected since the task monitor would have to be recompiled for each instance due to strong type checking. Task simple names cannot be used because they may not be unique. EDEN therefore assigns its own run-time identifier. Different deadlocks that are detected include:

- (1) Self-Blocking: check to see if a task has called itself.
- (2) Circular-entry-call: examine the entry-calling-graph of the program (which is a directed graph). When an entry call from task T1 to task T2 occurs, EDEN checks to see if the insertion of the edge $\langle T1, T2 \rangle$ would make a cycle in the graph. If so, circular deadlock has occurred.
- (3) Dependence-blocking: when task T1 makes an entry call on task T2, EDEN examines their dependency. If T1 is dependent on a block in the body of T2 or a subprogram called by T2, then dependence-blocking has occurred.
- (4) Global tasking communication deadlock: this is detected when the number of active tasks equals the number of blocked tasks.

Note that EDEN has been implemented, and at the time of the article was undergoing improvement.

German [Germ82] illustrated methods for the transformation of program P into P', with all of the imbedded monitor calls visible; the program was the dining philosophers. In a later work [Germ84], he illustrated the transform of program P into program P', which can experience deadlock iff P does also. When P' experiences deadlock, it can signal its occurrence. For producing unique task identifiers, German creates a unique integer for each task. The actual variable is stored local in the task's body. He was quite remiss about the fact that there is no good way to generate a task name. He suggested that the attribute t'taskname be added to the language.

To detect a circular deadlock, the transformed program is dynamically represented by a directed graph (V,E) with vertex V being for the tasks, and edges in E, represented by (t1,t2), indicate when task t1 has initiated an unanswered entry call on task t2. The graph can be modified by:

- (1) adding a new vertex (task startup),
- (2) adding a new edge (task t1 calls on task t2),

- (3) removing an edge (task t2 complete rendezvous with task t1), and
- (4) removing a vertex and all associated edges (the task that the vertex represents has terminated).

In the above he defines deadlock as follows: "A vertex in a state graph g is deadlocked (for the simple state model) iff it has an outgoing edge and there is no sequence of permissible transitions of g which leaves the vertex without an outgoing edge." Also: "a vertex in g is deadlocked iff there is a cycle reachable from it."

It is a common problem that a task cannot be properly monitored if it engages in any tasking activity during the elaboration of its declaration. German [Germ84] suggests modifying the program P so that the declaration is moved into an inner block, and thus statements can be executed before the elaboration that allow the monitor to be prepared for the elaboration.

Falis [Fali82] designed and implemented an Ada run-time task supervisor. His article discussed Adam, an Ada modification. It has removed inherent tasking, making it very low level. The site task scheduler is replaced by a run-time task supervisor package.

LeDoux [LeDou85] called a monitor to save "traces." A trace is a Prolog language clause that is later analyzed within a Prolog environment. Her technique used an "interval-based temporal logic approach." Program actions were viewed as events that appear to occur instantaneously, whereas program states are conditions that span a time interval. The system employed, called YODA, parses an Ada program, generates a symbol table, and outputs a transformed program that has inserted diagnostic output statements. The transformed program is then executed. Prolog clauses generated include the following:

entry_called()	var_updated()
call_canceled()	entry_parm_set()
entry_queue_lengthened()	task_activated()
entry_queue_shortened()	task_completed()
rendezvous_started()	ready_to_terminate()
rendezvous_completed()	program_ended()
var_read()	abnormally_terminated()

The location of the occurrence is identified by the program unit and the block ID (which is generated if it doesn't exist).

For variables, only scalars are supported. Entry families are not supported. A time stamp is given to each tasking occurrence. Prolog is used to interpret the results (asking such questions as "Which tasks updated X?"). The sample included in the article shows how it can be detected when tasks access/update a shared variable at the same time.

A paper by Gait [Gait86] goes over what is called the probe effect in concurrent programs. By introducing delays into the program, scheduling can be simulated. If the program's results seem to change based on the delays, then there may be synchronization errors in the program that make the program's results dependent on the way in which it is executed.

2.3.6 TESTING CONCURRENT PROGRAMS

While the entire purpose for the groundwork presented above is the actual testing of concurrent programs, it is clear for the literature that little has made its way into practice at this point. Tai [Tai85] presents a graphical notation for testing concurrent programs; however, his treatment is quite esoteric. Goldszmidt [Gold89] presented a black box approach toward testing programs written in concurrent languages. Hsuesh [Hseu89] concentrated more on data oriented debugging for concurrent programming languages. Also involved with debugging was Brindle [Brin89], who showed considerable insight into the problems involved in testing/debugging. LeDoux's approach [LeDou85] of saving traces appeared to be one of the most creative, especially as it relates to the past experience within QUEST. Also, Stone's [Ston88, Ston89] use of the concurrency map representation might be useful for depicting the structure of tasking events and for showing the "replay" of a tested tasking program (see Section 4 of [Ston89] paper). The floating nature of the concurrency map could also be employed by the "task scheduler/ monitor" in an attempt to force certain tasking progressions.

2.3.7 OPTIMIZATION OF ANALYSIS

Taylor [Tayl88] introduces methods that can cut down on the huge time-space requirements to perform static analysis or symbolic execution. One of the techniques is parceling. The basic static representation of all possible concurrency histories assumes that all tasks are active at the same time. This might not be true, and the sample space may be reduced significantly if it can be identified when tasks are inactive and thus cannot be considered as eligible for state transition. If tasks can be identified as being independent, they can be analyzed separate from the whole.

The following approaches were found for limiting computation explosions [TAYL83, TAYL88]:

1. Parceling of the analysis. The run-time for concurrency analysis of a large program with T tasks and n flow graph nodes per task is $O(n^T)$. The basic idea of parceling is to note when certain tasks are active, and consider these tasks only when they are needed rather than assuming that all of the

tasks are active at the same time. Parceling has the disadvantage of placing restrictions on the program.

2. **Weak monitors:** The use of a weak monitor group (example procedures, tasks, and packages) whose composition is to be applied to the program under analysis was suggested as a means to reduce computation. Weak monitors have the problem that they do not detect existing erroneous error states.
3. **Heuristic Search:** A heuristic function is defined as a "reasonable estimator of the distance (number of state transitions) between a given node and some node representing an error." The use of such a function to drive the search process is called a heuristic search. As an alternative to parceling and weak monitors, it does not have their inherent disadvantages. The heuristic search relaxes certain constraints on the concurrency state generator.

Taylor also provides methods to control generation of the symbolic execution graph.

2.3.8 CONCLUSION

In summary, the literature review has clearly revealed that a run-time monitor, possibly with task scheduling capabilities, is a major concept which should be integrated into the design of QUEST/Ada. Ideally, static analysis of concurrent tasks provides a wealth of understanding on the potential for tasking errors. Unfortunately, static analysis is expensive to perform on complex tasking programs. If, however, in practice the amount of tasking used is simple and easily managed, static analysis can be used to provide a potential concurrent history space to compare actual executions of the concurrent tasks against.

Task monitoring is essential in studying concurrent tasks. This requires transformation of the original program into a new program that calls the task monitoring prior and after tasking activities. The task monitor, upon the main program's impending termination, can save the tasking information to storage. This information represents a concurrent history of one instance of execution. The monitor can also dynamically find when shared variables are updated in parallel and when deadlock is about to occur in the tasking programs.

The monitor can be augmented by a simple scheduler that attempts to force the tasking program through a predetermined path of concurrent execution. This would be most useful if static analysis were used to produce the potential concurrent history space. Each proper concurrent history in the potential space could then be attempted, and if successful (as noted by the output of the monitor) that history would be checked off as covered.

The issue of test data is complicated by concurrency. In addition to path coverage, concern must be with concurrent history coverage. If static analysis is available, all potential concurrent histories can be generated. The output of the monitor task, a true concurrent history, can be compared against the potential concurrent history space, and the matching member of the potential space can be checked off. The remaining members in the potential space are goals for execution. Test data cannot be executed with confidence for one instance of a concurrent history since the program might produce different results for the same set of data when executed through different concurrent histories.

The main advantage of concurrency analysis is that it provides insight into the tasking interactions with concurrent programs. The major errors that the analysis purports to find -- rendezvous deadlock and shared variable parallel update -- would not occur in the Ada program that uses Ada's advanced tasking features that were especially designed to avoid these problems. By using the monitor task and by examining the potential concurrent histories, any tasking logic errors, however, can be identified.

3.0 PROTOTYPE DEVELOPMENT

3.1 OVERVIEW OF THE QUEST/ADA PROTOTYPE

One important purpose of the QUEST/Ada project is to determine the viability and effectiveness of the rule-based testing paradigm. In order to collect data to determine the effectiveness of this approach, a prototype of the QUEST/Ada system has been developed. This prototype consists of five parts, which are discussed briefly below. Each will be described in greater detail in the subsections which follow this one.

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

Once the source module has been scanned and instrumented, initial test data are prepared for it by the Test Data Generator

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

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

The Symbolic Evaluator (SE) uses extremely detailed information about the source code being tested to attempt to represent each path through the code as an abstract function. The work of the symbolic evaluator is divided into two parts -- developing and evaluating symbolic expressions. Using descriptions of the conditions in the module under test provided by the PSM, the SE develops symbolic boundary expressions in which each of the variables in a condition is represented in terms of the other variables. This boundary expression describes the point at which the control variable will cause the Boolean condition to evaluate to equivalence. Thus, by adding or subtracting a small value to the boundary (called epsilon), the Boolean inequality can be forced into each of its three states. After developing the symbolic boundary equations, the SE evaluates them using the test data as it appears at the time the condition is executed. In mathematical terms, if $D_i(t)$ is the input test data, $D_c(t)$ is the value of the variable at the condition in question, and $D_b(t)$ is the boundary

value for that variable at that condition, then a simple abstract function heuristic might select $D_i(t+1) = D_b(t) * (D_i(t) / D_c(t))$.

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

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

3.2 TEST DATA GENERATOR

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

3.2.1 BEST TEST CASES

The objective of the Test Data Generation (TDG) component of QUEST is to achieve maximal branch coverage. In order to assure the direction of test case generation to be fruitful, a branch coverage analysis is needed. The coverage analysis of this framework follows the Path Prefix Strategy of Prather and Myers [PRA87]. In this strategy, the software code is represented as a simplified flow chart. The branch coverage status of the code is recorded in a coverage table. When a branch is driven (or covered) by any test case, the corresponding entry in the table is marked with an "X". Figures 3.2a and 3.2b indicate a sample flow chart and its coverage table. The goal of the test case generation is to fill all the entries in the table, if possible.

The coverage table provides not only information regarding the branches covered but also direction for further test case generation. Consider Figures 3.2a and 3.2b. Currently, conditions 1 and 2 are fully covered; conditions 3, 4, and 5 are partially covered; and condition 6 is not covered. Since conditions 1 and 2 are fully

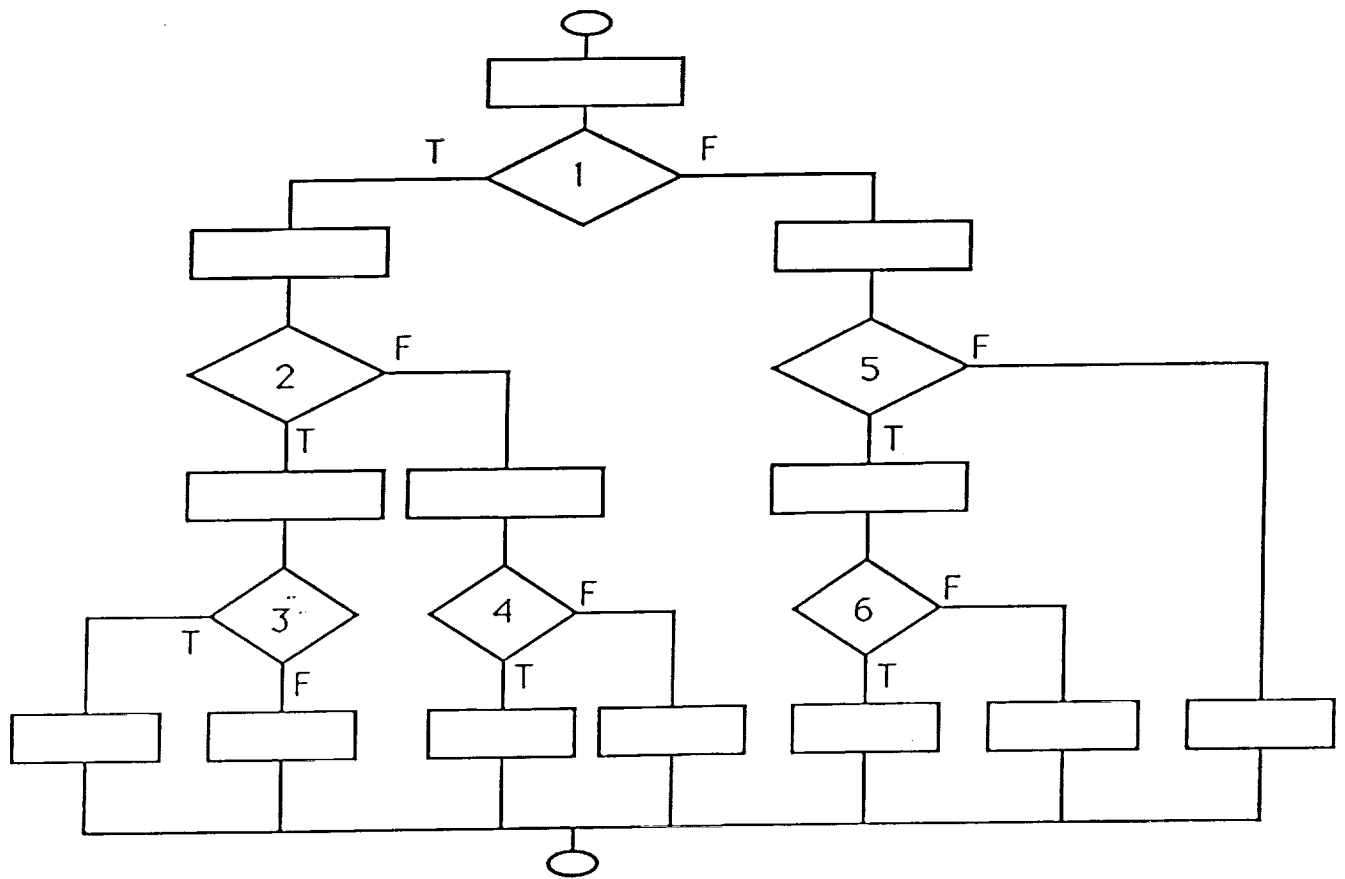


Figure 3.2a A sample flow chart

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

Figure 3.2b Coverage table of Figure 3.2a

covered, there is no need to generate more cases to cover them. Condition 3, on the other hand, is partially covered. More cases should be generated to drive its false branch, i.e., 3F, which is not yet covered. The Path Prefix Strategy states that new cases can be generated by modifying a test case, say case 3T, that has driven 3T. Consider the fact that case 3T starts at the entry point and reaches condition 3. Although it drives 3T, it is "close" to driving 3F. Slight modification of case 3T may devise some new cases that will drive 3F.

With this strategy in mind, the test case generator should target partially covered conditions. Earlier test cases can be used as models for new cases. Conditions that have not been reached yet, e.g., condition 6 in Figure 3.2b, will not be targeted for new case generation. This is because no test case model can be used for modification. A model will eventually surface later in the process. In this example, after condition 5 is fully covered, a model for condition 6 will appear.

Problems arise when there is more than one test case driving the same path. For example, if cases 1, 2, ..., n all drive branch 3T of Figure 3.2b, then the selection of the case to be used as the model for branch 3F becomes problematic. If all cases are used, efforts are likely to be duplicated, which is not efficient. Since an automatic case generator can generate a large amount of cases, it would be necessary to quantify the "goodness" of each case and use the "best" case as the model for modification.

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

Consider the typical format of an IF statement: IF exp THEN do-1 ELSE do-2. The evaluated Boolean value of exp determines the branching. Exp can be expressed in the form of: lhs <op> rhs. Lhs and rhs are both arithmetic expressions and <op> is one of the logic operators such as <, >, <=, >=, <>, and =. The goodness of a test case, t1, relative to a given condition can be defined as

$$|lhs(t1) - rhs(t1)| / \text{MAX} (|lhs(t1)|, |rhs(t1)|) \quad (1)$$

Lhs(t1) and rhs(t1) represent the evaluated value of lhs and rhs, respectively, when t1 is used as the input data. This measure tells the closeness between lhs and rhs [DEA88]. When this measure is small, it is generally true that a slight modification of t1 may change the truth value of exp, thus covering the other branch. The importance of slight modification to a model test case is based on the fact that the model case starts from the entry point and reaches the condition under consideration. Between the entry point and the condition, the modified cases must pass through exactly the same branching conditions and yield the same results. For this

reason, the smaller the modification is, the better the chance will be for a modified case to stay on the same path [PRA87]. The given closeness of lhs and rhs provides a way of measuring this goodness.

The goodness measure of (1) may range from 0 to 2. It can be normalized so that the measure will range from 0 to 1. This is done by dividing equation (1) by 2. The new definition will be

$$|lhs(t1) - rhs(t1)| / (2 * MAX(|lhs(t1)|, |rhs(t1)|)) \quad (2)$$

With this definition, a test case that yields the smallest measurement is considered to be the best test case of the condition under consideration.

The closeness measurement of (1) and (2) has a serious risk, however. Recall that a set of new test cases is generated based on the best test case of a partially covered condition (called target condition), and the intent of the new test case set is to cover the uncovered branch of the target condition. Although we define the slighthness of modification of a test case as its goodness, this measure is computed based on the target condition only. A slight modification to the lhs and rhs of the target condition may not have the same meaning to those conditions on the path. This may result in unanticipated branchings along the path, therefore losing the original purpose of the new cases. In order to reduce the likelihood of unanticipated branching, a test case's goodness measure should also consider those conditions that are on the path. This idea can be expressed in the following example.

In Figure 3.2.1a, two test cases, t_a and t_b , pass through the false branches of conditions D_1 , D_2 , and D_3 . Assume the current effort is to generate more cases such that the truth branch of D_3 will be covered. Either t_a or t_b should be used as a model for the new cases. If the whole input space is represented as R , the input space can be divided into several subspaces (see Figure 3.2.1b). First, R is divided into $1T$ and $1F$, which represent the portions of input space that drive the truth and false branches of D_1 respectively. Similarly, $1F$ can be divided into $2T$ and $2F$, and $2F$ can be divided into $3T$ and $3F$.

In this example, both t_a and t_b fall within the subspace of $3F$. If we want to drive the other branch of D_3 , new cases should come from the subspace of $3T$. A best test case must be selected between t_a and t_b . According to the earlier definition, goodness is the distance that each test case is from the boundary of $3T$ and $3F$. Based on this definition, t_a is closer to the boundary so it is chosen as the better test case. From the viewpoint of D_3 this is correct. A relatively small modification to t_a may lead to $3T$. However, t_a is also close to the boundaries of D_1 and D_2 , so there is a good chance that a slight modification to t_a may lead to undesired branches at D_1 and D_2 .

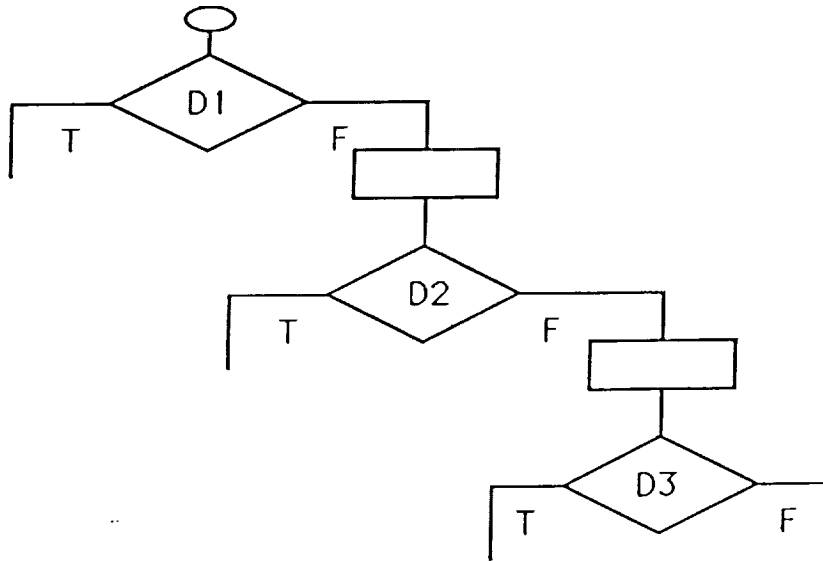


Figure 3.2.1a A sample program

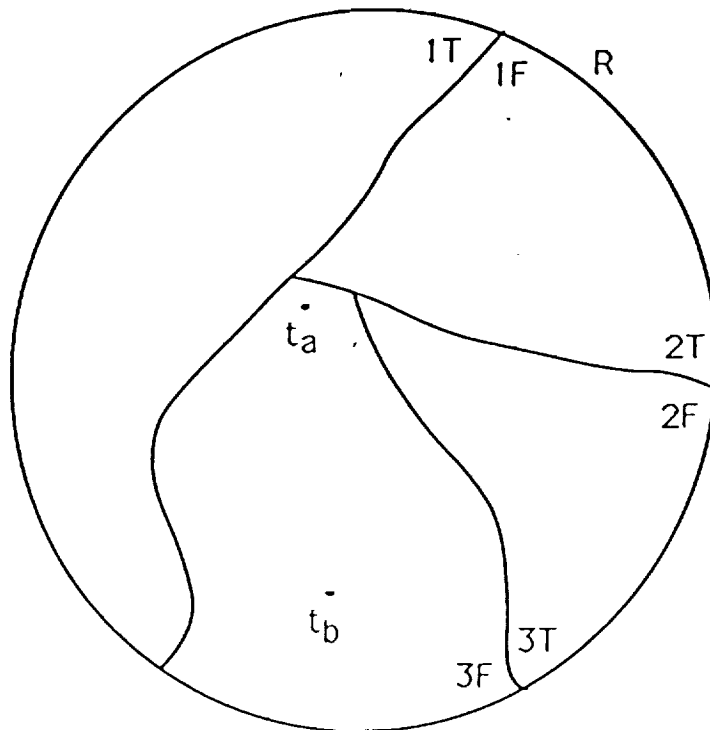


Figure 3.2.1b Input space of the program in Figure 3.2.1a

We will call the magnitude of modification that is required to drive a different branch at a condition the freedom space of a test case. In this example, t_a has a small freedom space at D_3 which is desirable. But its freedom spaces at D_1 and D_2 are also small, which may cause unanticipated branchings. On the other hand, although t_b is not as close to D_3 's boundary as t_a is, it is not close to any other boundaries either. A larger modification may be required for t_b to lead to 3T. Since t_b is far away from any other boundaries, a larger modification may not cause any unanticipated branches. For this reason, the goodness of a test case concerning a particular condition should be determined by the freedom space at the target condition as well as the freedom spaces of all conditions that are on the path to the target condition. For the former element, the smaller the better; for the latter element, the larger the better. The goodness can now be redefined as:

$$G(t,D) = w * L(t,D) + (1-w) * P(t,D) \quad (3)$$

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

$L(t,D)$ is defined as 2, and $P(t,D)$ is defined as:

$$P(t,D) = \sum_{D_i} 1 / (n * L(t,D_i)) \quad (4)$$

Here, D_i is a condition that is on the path toward D , and n is the total number of these conditions. Although this definition does not represent the actual distance of test case t to a boundary, it is a reasonable approximation. According to this definition, the smallest value indicates the best test case.

Although formula (3) seems more appropriate than formula (2), in terms of test case goodness measurement, it would be difficult to prove it theoretically, since both definitions are derived heuristically.

When a test case is run in the test case analyzer and it reaches a condition that is either partially covered or not covered at all, its goodness value is computed. This value is then compared with the goodness value of the current best case, if there is one. If its value is smaller, this test case replaces the original case and becomes the new best case. In the implementation, the test case analyzer actually keeps more than one test case for each partially covered condition. That is, the second, the third, and the fourth best cases are also kept. This provides

alternatives for the test case generator when the original model does not yield new coverage.

3.2.2 TEST DATA GENERATOR PROCEDURE

When a new test case is generated, it is intended to cover a particular branch. This intended branch always belongs to a partially covered condition, except in the very beginning of test case generation. Based on the best test case of a targeted partially covered condition, a slight modification to the case is made with the intent to lead the execution to the uncovered branch of the target condition. The importance of "slightness" is to keep the new test case following the original execution path with the exception resulting in the target condition. The main issue in the research has been the establishment of methods for efficiently performing this modification.

Consider Figure 3.2.2. Input to the procedure contains three parameters x , y , and z . Assume condition D is partially covered and its best test case is (x_1, y_1, z_1) . We try to generate more cases to cover D 's false branch. Condition D can be expressed as $\text{lhs}(x, y, z, v_1, v_2, \dots) <\text{op}> \text{rhs}(x, y, z, v_1, v_2, \dots)$. Here, v_1, v_2, \dots are internal variables of the procedure. Input parameters x , y , and z may or may not be modified between the entry point and condition D . In this case, if (x_1, y_1, z_1) is input into the procedure, the evaluation of D will result in a truth value. What we are trying to accomplish is to modify (x_1, y_1, z_1) such that the evaluation of D will be false. The following sections discuss some heuristics that can be used to generate new cases.

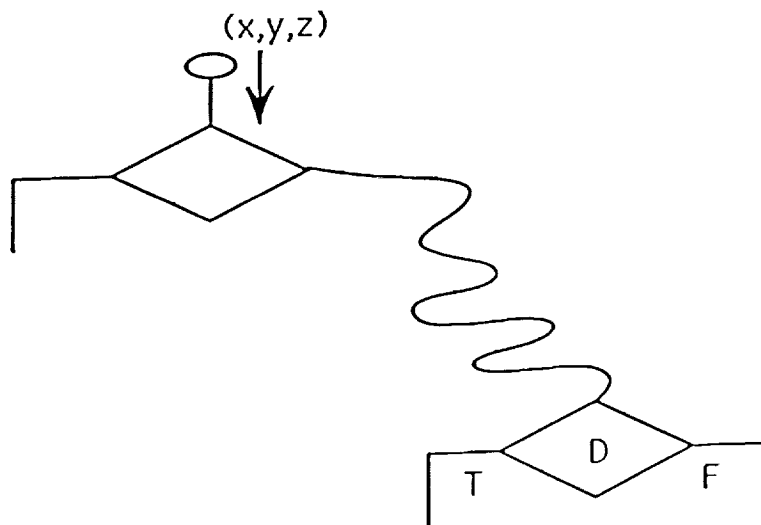


Figure 3.2.2 A test case (x, y, z) drives condition D

3.2.2.1 FIXED PERCENTAGE MODIFICATION

One way of generating new cases is to modify each parameter of the best test case with a fixed percentage of each parameter's ranges. The percentage can be any one of or any combination of 1%, 3%, 5%, 10%, etc. For example, if the best test case is (x_1, y_1, z_1) and the ranges for input variables x , y , and z are $[0 \ 10]$, $[-100 \ 0]$, and $[-50 \ 50]$ respectively, a 1% modification would generate two new cases. They are $(x_1+0.1, y_1+1, z_1+1)$ and $(x_1-0.1, y_1-1, z_1-1)$. Several different combinations can be used at the same time. This would provide more new cases. After a new case is generated, it must be checked to ensure that each variable is within its range.

3.2.2.2 RANDOM MODIFICATION

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

3.2.2.3 MODIFICATION BASED ON CONDITION CONSTANTS

This method generates new cases based on the constants appearing in a condition. Depending on the number of constants in a condition, different rules can be applied. For example, if there is one constant and one input variable in a condition, then generate a new case by putting the constant in the position of the input variable in the best test case. This rule is designed for conditions of the form: $x <op> C$, where C is a constant. Similarly, for two constant conditions, e.g., $x+C_1 <op> C_2$, three new cases can be generated. They are C_1+C_2 , C_1-C_2 , and C_2-C_1 . Rules for conditions with more constants have similar forms. These rules were developed by DeMillo, Lipton, and Sayward [DEL78], and Howden [HOW87], who are considered to be experts in software test case generation. Implementation of this kind of heuristic has been reported in a separate paper [DEA89], in which these rules are represented in Prolog. Performance of this approach shows a significant improvement over randomly generated test cases.

3.2.3 TEST CASE GENERATION RULE ORGANIZATION

To date, this research has developed many test case generation (TCG) rules. It may not be desirable to use them all at once, since too many unwanted cases may be generated. Sometimes one case covering a particular branch would satisfy the coverage requirement, and extra cases are simply a waste of effort. In this

situation, test cases can be generated in an incremental manner. That is, TCG would stop when a predefined criterion is met. On the other hand, multiple cases covering a particular branch provide a larger pool for best test case selection. The purpose of this section is to present an initial organization of the TCG rules. If it is found to be desirable to keep the number of test cases down, then the following rule organization scheme can be applied.

Associated with each best test case, define a numeric flag, FG, set to 1 initially. Every time a best test case is used for case generation, its FG is incremented by one. The test case generation rules are divided into groups. When more cases are to be generated, FG is used as an index to the rule groups. This guarantees that a different rule group will be used for a given best test case in each loop. This will avoid repetition and wasted effort. This scheme is expressed below:

CASE GENERATION FOR CONDITION-i

1. Retrieve best-test-case, BTC-i, of CONDITION-i;
2. $N = \text{FG of BTC-i}; \text{FG} = \text{FG} + 1;$
3. Select and apply the N-th rule group;
4. Test run and analyze new cases;
5. IF no-new-coverage-is-achieved
6. THEN IF rule-group-is-not-exhausted
7. THEN goto step 2
8. ELSE no-additional-coverage-can-be-achieved-by-BTC-i
9. ELSE CONDITION-i-is-fully-covered;

Note that in step 4 a new best test case may be defined. In that situation FG would be reset to 1. Recall that a target condition is already partially covered. Any new coverage will lead to full coverage, i.e., step 9. However, if the rule groups have been exhausted before additional coverage can be achieved, something else must be done, i.e., step 8. This is further discussed in the next subsection.

One example of organizing the rule groups follows:

GROUP-1

- a. Modify single variable through symbolic manipulation.
- b. Modify single variable by 1%, and 5%.

GROUP-2

- a. Modify two variables - one variable is bound to its mid-range and the other is computed through symbolic manipulation.
- b. Modify single variable by 10%, 20%, and %50.

GROUP-3

- a. Modify three variables - two variables are bound to their mid-ranges and the third one is computed through symbolic manipulation.
- b. Modify two variables by 2%, 10%, and 20%.

While these examples demonstrate one potential rule group organization, the test case generation rules will not be limited to these, i.e., many alternatives will be tried in order to improve the performance of the test case generator.

3.3 PARSER/SCANNER

3.3.1 BASIC INSTRUMENTATION

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

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

LHS <relational operator> RHS.

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

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

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

The functions are encapsulated in Ada GENERIC packages to facilitate parameter passing and input/output of user-defined types.

Currently, packages are available for integer, enumerated, floating point, and fixed point data types.

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

3.3.2 INSTRUMENTATION FOR SYMBOLIC EVALUATION

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

3.3.3 INSTRUMENTATION FOR MULTIPLE CONDITIONS

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

IF (a < b AND c > d) THEN

would be translated to the following statement:

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

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

3.3.4 AUTOMATIC INSTRUMENTATION

The instrumentation described here is currently being performed manually. Although automatic instrumentation could be performed during the execution of the Parser/Scanner Module, its

implementation would require considerable effort which would greatly hinder progress on the other substantial areas of the research and prototyping. Also, the form of instrumentation is currently in a state of flux and will probably not stabilize in the near future. This is seen to be a relatively straightforward task for those in the industry who are specializing in the design and development of Ada compilers. In fact, this could be integrated into the compiler and debugger tools in a very efficient manner. For these reasons, it was decided that prototyping of the automatic instrumentation would not be pursued immediately. However, the requirements for automatic instrumentation become quite apparent from the manual examples which are being employed to test the remainder of the QUEST system. Examples of instrumented programs and source code for the instrumentation packages may be found in Appendix 3.

3.4 COVERAGE ANALYZER

In order to experiment with the effects of altering the knowledge about the conditions of a program under test, three categories of rules have been selected. The first category of rule reflects only type (integer, float, etc.) information about the variables contained in the conditions. These rules generate new test cases by randomly generating values. As implemented, these rules determine lower bounds, higher bounds, and types of the variables. A random value of the type is generated, and the value is checked to be sure it is within the range for the variable.

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

The final type of rule utilizes information about the condition that can be obtained by symbolic manipulation of the expression. The given rule uses a boundary point for input variables associated with the true and false value of a condition. This value is determined by using symbolic manipulation of the condition under test. Many values can be chosen that cross the boundary of the condition and, as with best test case selection, a value is sought that will not alter the execution path to the condition. In addition to best test case selection, this rule base has additional knowledge to generate new test cases. The values of variables at a condition are compared with input values of the

variables used to reach that condition. This added information is incorporated in the generation of new test cases.

Suppose that for an input variable x appearing in a condition under test, the value of x at the condition boundary has been determined to be x_b and the input value that has driven one direction of the condition has been x_i . We do not know how x is modified along the path leading to the condition since the value of x on input may be expected to differ from the value of x at the condition. However, we are able to establish that the value of x at the condition is x_c . Provided the values lie in the limits allowed for values of x , the new test case is chosen as:

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

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

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

3.4.1 AUTOTEST AND THE TEST COVERAGE ANALYZER

The purpose of the Autotest module is to coordinate the activities of the Test Data Generator (TDG), the module under test, and the Test Coverage Analyzer (TCA). Autotest repeatedly calls the above procedures until all of the required test packets are complete. The test data generator and the module under test are covered elsewhere (Sections 3.2); the TCA is described below.

The primary job of TCA is to supply the TDG with the best test cases which have been used to execute the module under test. It also accumulates data for reports after the test and archives results of the test.

A best test case is chosen for each condition in the module under test. There can be several different methods for choosing the best test case. Currently, two methods have been implemented. The first is to calculate the distance each test case is from a border of the condition, and then select the case which is closest to the border. For instance, if the condition is

$$x * 3 < 15$$

then the border is at $x = 5$, and that condition with test data that produces a value of x closest to 5 is considered the best test case.

The second method for choosing a best test case involves the above procedure augmented by steps for the avoidance of previously encountered conditions. In this approach test cases are selected for closeness to the current condition and distance from all of the previous conditions. The methods for selecting the best test cases are more fully described below.

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

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

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

```
for each test packet
    call the TEST_DATA_GENERATOR
    call the MODULE_UNDER_TEST using the test data
    call the TEST_COVERAGE_ANALYZER.
```

The following logic is used by the TCA module:

```
for each intermediate results record
    calculate the "goodness" values of the test case
    if the condition is not in the coverage_table
        install the condition
    else
        if the condition is not fully covered
            update the condition using "goodness" values
for each condition in the coverage_table
```

if the condition is partially covered
return its best test cases to the TDG
accumulate data for test reports
archive the results.

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

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

This section continues with subsections which extend these basic concepts to decisions which involve multiple terms.

3.4.2 TEST CASE GENERATION FOR COMPOUND DECISIONS

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

3.4.2.1 TEST CASE GOODNESS MEASURES

If a condition contains Boolean variable(s) only, the test case goodness measure should be based on the sum of condition boundary closeness along the path leading to the target condition. Since only Boolean variables are involved, closeness measurement cannot be done at the target condition. However, if there is at least one arithmetic expression in the condition, a normalized boundary closeness measure can be used. For example, given a test case, (x=12, y=-8, and z=8), and a statement, IF (x >= 10) OR (y <= -10) THEN do-1 ELSE do-2. The boundary closeness measure of

each individual term can be calculated first. For the first term, $(x \geq 10)$, the measure is $|12 - 10| / (2 * \text{MAX}(|12|, |10|)) = 2/24$; for the second term, $(y \leq -10)$, the measure is $2/20$. The normalized measure is simply the average of these two measures. After this point, earlier definitions of goodness can still be applied.

3.4.2.2 TEST CASE GENERATION RULES

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

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

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

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

Once the assignment to each condition is determined, test cases must be generated to satisfy the requirement of each condition. Unfortunately this may involve solving a set of predicates which has been recognized as a difficult problem, as referenced above. In order to simplify the test case generation, the following heuristic rules will be tested:

RULE-1:

IF a condition contains Boolean variables only
THEN invert the values of those variables appearing in the input list of the best test case, one at a time.

RULE-2:

IF a condition contains no Boolean variable
THEN consider each Boolean term individually and sequentially;
first find the boundary, then generate cases around the boundary.

RULE-3:

- IF a condition contains both Boolean variables and non-Boolean terms
- THEN
1. invert the values of the variables appearing in the input list of the best test case, one at a time, and
 2. consider each Boolean condition individually and sequentially; first find the boundary, then generate cases around the boundary.

These preliminary rules may not generate cases to cover all desired branches, but they will serve as the beginning for multiple condition test case generation.

3.5 SYMBOLIC EVALUATOR

3.5.1 BOUNDARY COMPUTATION

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

$$\begin{aligned}x_b &= f1(y, z, v_1, v_2, \dots) \\y_b &= f2(x, z, v_1, v_2, \dots) \\z_b &= f3(x, y, v_1, v_2, \dots)\end{aligned}$$

This set of expressions defines the condition boundary of D for x, y, and z. They can be derived from D using symbolic manipulation. For example, if we have a condition

$$x + 3 * y \leq 4 - 6 * z + v$$

The condition boundary will be

$$\begin{aligned}x_b &= 4 - 6 * z + v - 3 * y \\y_b &= (4 - 6 * z + v - x) / 3 \\z_b &= (4 - x - 3 * y + v) / 6\end{aligned}$$

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

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

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

```
IF     $x_1 = x_c$ 
THEN generate three new cases
       $(x_b, y_1, z_1)$ ,
       $(x_b+e, y_1, z_1)$ ,
       $(x_b-e, y_1, z_1)$ .
```

Rules for other input variables would have the same form.

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

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

$$|x_b - x_c| / (2 * \text{MAX}(|x_b|, |x_c|)) * 100 \% \quad (5)$$

Formula (5) is identical to (2) but has a different interpretation. Following this measurement, we can modify input x with the same percentage. One more question needs to be answered: how should the percentage of x be defined? For example, if we want to modify x by 12% and $x_i = 10$, the answers should not be simply 11.2 or 8.8. This is because the input space of x may be something like $[-1000, 200]$. Percentage based on x_i may not reflect the input space of x at all. The proposed calculation is to use the input range size of x , i.e., $[\text{upper_limit_of_}x - \text{lower_limit_of_}x]$, as the basis. In this example, the range size of x is $200 - (-1000) = 1200$, and the new boundary values for x would be $10 + 144 = 154$ or $10 - 144 = -134$. The values of x for new test cases should result in conditions which are slightly off the boundary as well as those right on the boundary. If we use one percent of x 's range as the variation, i.e., $e = 12$, six new cases can be generated. While all other variables remain unchanged, new values for x will be 142, 154, 166, -146, -134, and -122. This heuristic can be integrated into the earlier rule to yield:

```

IF  $x_i = x_c$ 
THEN generate three new cases           ;no modification
    ( $x_b, y_1, z_1$ ),
    ( $x_b + e, y_1, z_1$ ),
    ( $x_b - e, y_1, z_1$ ).
ELSE compute boundary value,  $x_b$ ,       ;modification along path
    compute off target percentage using (5),
    approximate input boundary values using input range,
    generate new cases for being on or slightly off boundary.

```

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

In this section, several heuristic rules have been presented. It is likely that each rule is effective in certain situations. If several rules are applied to a program, they will complement each other and yield better coverage.

3.5.2 FACTS USED BY THE SYSTEM

The rules accept the following three types of facts:

1. (names var_name1 var_name2 ... var_namen)
 where var_namei are the names of variables accessible to the module;
2. (val-at-cond test_num decision_num condition_num var_value1 var_value2 ... var_valuen)

where var_valuei are the values of the variables at the point of this decision and condition for this test case data; and

3. (cond-expr decision_num condition_num conditional_expression)

where the conditional_expression is in fully parenthesized infix notation.

Using these facts they generate the following intermediate facts while working:

1. (number-of-variables ?n)

is used to build the correct-length list-of-nils.

2. (list-of-nils NIL NIL ... NIL)

is used later to initialize the boundary-values to NIL.

3. (lhs ...)

4. (rhs ...)

5. (variable ?x)

6. (working decision_num condition_num ?x)

7. (number-of-variables-done ?n)

8. (boundary-expr decision_num condition_num boundary_expression)

9. (evaluate test_num decision_num condition_num boundary_expression)

Items 3-7 are all used during the symbolic manipulation of expressions to produce the boundary-expressions. Items 8-9 are used to find the boundary-values. The list-of-nils and boundary-expr facts are retained for use with other test cases.

The final result is the assertion of boundary-values facts (one for each test_num, decision_num, condition_num combination) of the form:

(boundary-values test_num decision_num condition_num var_value1 var_value2 ... var_valuen)

where var_valuei are the boundary values for the variables in the decision-condition expression for this test case. Boundary values are found by solving the expression symbolically for the variable of concern and substituting the val-at-cond values for the remaining variables. Variables not present in the expression are given a boundary value of NIL.

3.5.3 SALIENCE LEVELS OF RULES

Salience levels are used in the Clips language to force a required preordering among groups of rules. A rule will not execute until all rules of higher salience level have executed. The following salience levels are used in the Symbolic Evaluator:

```
100  swap-right-and-left
      do-not-swap-right-and-left
0    rules to manipulate symbolic expressions
      initialize-empty-list-of-nils
      build-list-of-nils
      variable-not-in-condition
-50  assert-boundary-expr
      crash-and-burn
-100 incrementer
      cond-expr-done
-150 start-one
-200 prepare-for-evaluation
      substitute
-250 evaluate
      set-up-null-boundary-values
-300 assert-boundary-values
```

3.5.4 CONTROL FLOW

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

```
initialize-empty-list-of-nils
build-list-of-nils
while not all cond-expr's done
  start-one (prepare to solve a cond-expr for first variable)
  while this cond-expr not done
    [do-not-]swap-left-and-right (get variable on left side)
    if "variable" is not in cond-expr
      then variable-not-in-condition
    else solve expression for variable
      assert-boundary-expr
  if condition not successfully simplified
    then crash-and-burn
  if this cond-expr is done (solved for all variables)
    then cond-expr-done
```



```

        else incrementer (prepare to solve cond-expr for next
                           variable)
    for each combination of "val-at-cond" and "boundary-expr" facts
        prepare-for-evaluation (set up "evaluate" facts)
        substitute ("val-at-cond" values for variable-name)
    while not all "evaluate" facts fully reduced
        evaluate (reduce right hand side arithmetically)
    for all "val-at-cond" facts
        set-up-null-boundary-values (initialize to list-of-nils)
    for each simplified "evaluate" fact, i.e. boundary value
        assert-boundary-values (replace NIL with actual value)

```

3.5.5 AN EXAMPLE

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

```

initial-fact
* initializes the fact list.
names x y z q abba v
* list of variables in this module.
val-at-cond 0 0 0 T 1 2 3 4 5 6
* value of the variables at Test 0, Condition 0, Decision 0.
val-at-cond 0 1 0 T 1 2 3 4 5 6
val-at-cond 1 1 0 T 10 20 30 40 50 60
val-at-cond 1 0 0 T 10 20 30 40 50 60
cond-expr 0 0 "(" x "+" "(" 3 "*" y ")" ")" "<=" "(" 4 "-" "(" 6
6 "*" z ")" ")" "+" v ")"
* conditional expression - (x + (3 * y)) <= ((4 - (6 * z)) + v).
cond-expr 1 0 x "=" y

```

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

```

initial-fact
names x y z q abba v
val-at-cond 0 0 0 1 2 3 4 5 6
val-at-cond 0 1 0 1 2 3 4 5 6

```

```

val-at-cond 1 1 0 10 20 30 40 50 60
val-at-cond 1 0 0 10 20 30 40 50 60
cond-expr 0 0 "(" x "+" "(" 3 "*" y ")" ")" "<=" "(" "(" 4 "-" "("
6 "*" z ")" ")" "+" v ")"
cond-expr 1 0 x "=" y
* the original facts remain in the listing.
list-of-nils NIL NIL NIL NIL NIL NIL
* a NIL is generated for each variable.
boundary-expr 1 0 x "=" y
boundary-expr 1 0 y "=" x
* boundary expressions are generated for both the left and right
* side of the conditional expression. Note: The last "cond-
* expr" is evaluated first.
boundary-expr 0 0 x "=" "(" "(" "(" 4 "-" "(" 6 "*" z ")" ")" "+"
v ")" "-" "(" 3 "*" y ")" ")"
boundary-expr 0 0 y "=" "(" "(" "(" 4 "-" "(" 6 "*" z ")" ")"
"+" v ")" "-" x ")" "*" 0.33333334 ")"
boundary-expr 0 0 z "=" "(" "(" 4 "-" "(" "(" 3 "*" y ")" "+"
x ")" "-" v ")" ")" "*" 0.16666667 ")"
boundary-expr 0 0 v "=" "(" "(" "(" 3 "*" y ")" "+" x ")" "-" "("
4 "-" "(" 6 "*" z ")" ")" ")"
boundary-values 1 0 0 -176 -42 -1 NIL NIL 246
* boundary values are generated for "val-at-cond's" * ( t e s t
condition) 0 0, 1 1, 0 1, and 1 0.
boundary-values 1 1 0 20 10 NIL NIL NIL NIL
boundary-values 0 1 0 2 1 NIL NIL NIL NIL
boundary-values 0 0 0 -14 -3 0.5 NIL NIL 21

```

3.5.6 SYMBOLIC EVALUATOR INTERFACE

The Symbolic Evaluator requires the intermediate results from the Instrumented Code Generator and the conditional expressions from the Parser/Scanner in order to generate facts and then execute. The intermediate results and conditional expressions are put into files for the Symbolic Evaluator to read so that it can generate the required facts. The files are read, facts generated, and boundary results created. The files are then closed awaiting new intermediate results.

3.6 LIBRARIAN

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

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

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

3.6.1 BASIC CONCEPTS

A collection of data files contain binary records representing information that has been archived from QUEST. These data files are also known as "flat files" because the data files themselves are void of any indexing. Separate files exist to aid in indexing the data files. The name of an indexing file is the name of the data file concatenated by the key number that the index file represents. Key numbers start at zero (which is usually the unique key for the data file). For instance, if the file name was test1.dat, the index file name for key number zero would be test1.dat00, and the index file name for key number one would be test1.dat01.

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

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

Remember that the index files for the data files are the same except that the key number is tacked on to the end of the file name.

All of the routines return a result code. Basically, if the return code is below zero, an error has occurred. If the return code is zero, the function executed without any bothersome events. If the return code is greater than zero, some event has occurred which, although not an error, might be important information for QUEST users (an end of file, for example). All of the return codes

are established in the header file `librarian.h` by `#define` statements.

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

3.6.2 USING THE LIBRARIAN

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

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

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

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

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

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

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

3.6.3 DETAILS OF THE CODE

The librarian is designed to rely on another set of code to do the detailed work of creating indexes into a file. All the librarian routines do is take a binary collection of data and save it somewhere, leaving a method to quickly find the data again later. The librarian was first designed using VAX RMS, but this reduced portability. Therefore, the BPLUS collection of B-tree index file management routines were employed.

Any given binary data record must possess the following attributes:

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

The librarian routines use the data set number for an index to access a global structure called `lib_glbl`. This global structure is very important because it is used to store descriptive attributes about each active file. This includes record size, number

of keys, and the keys that have been set for the given record. Currently, lib_glbl is initialized in the function lib_b_setup(), which is called during execution of lib_set(). The keys for a record, although likely made up of components within the record, are not stored with the record in the data file. The function lib_set_key() needs to be called for each key in a record before the record is written out. Each time lib_set_key() is called, the associated key string in lib_glbl is updated.

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

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

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

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

3.6.4 BPLUS PORTABILITY NOTES

Much of the source code employed in the Librarian was originally intended for execution under MS-DOS. It was developed for the Microsoft C and the Borland Turbo C compilers. For the most part, standard C routines are employed for the file management. These routines, commonly known as the "UNIX" class of file routines, include open(), read(), write(), and close(). These routines should be standard in almost any implementation of a C compiler. Porting to the VAX required the deletion from the BPLUS.H and the BPLUS.C files of all instances of "cdecl" and of "Pascal". The #include statements had to be rearranged to either not include a file that did not exist on the VAX or to remove a "sys\" directory specification. Additionally, a filelength() function had to be written to allow the length of a file to be

determined given the file's descriptor number. A phony #define for `O_BINARY` has been added so that an `open()` call succeeds. This binary specification is required for MS-DOS and other compilers that default to character translation for their data files.

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

The `BPLUS.H` and `BPLUS.C` files contain function prototypes for the `BPLUS` functions. Only a compiler that contains the ANSI extensions to handle function prototypes can deal with their presence. Older style compilers (K&R vintage) will abort compilation on encountering the function prototypes, requiring the declarations to be modified in order for the program to compile. Only the arguments contained within the prototype declaration need to be removed.

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

4.0 EXPERIMENTAL EVALUATION

4.1 EXPERIMENTAL DESIGN

Fractional decision coverage (FDC) was used as an initial metric of test case quality; $0 \leq \text{FDC} \leq 1$. FDC is defined to be the fraction of decisions covered by all test cases tried to a given point in the testing process. The objective of the experimental design was to determine if FDC is significantly larger under strategy i ($i = 0, 1, 2, \dots, n$), where strategy 0 is random test case generation and strategies 1, 2, ..., n represent n versions of rule-based test case generations. Let the versions be arranged according to the timing of their design and development. Thus, it was expected that each version would produce an improved strategy, i.e., $\text{FDC}_i > \text{FDC}_{i-1}$ for all i . The determination if statistically significant differences existed between the FDCs for the various strategies formed the basis for the experimental design.

In order to determine a single value of FDC for each strategy it was necessary to fix certain parameters. Since FDC is a function of the number of test cases tried, it was necessary to fix the number of test cases generated, N , to the same value within each strategy. A reasonably good value of N to determine if

differences exist between alternative strategies was determined by early experimentation. In order to test the sensitivity to N , selected experiments were repeated at values of $N+50$ and $N-50$ and the results were compared to those obtained at value N . (Note: if N is found to exceed 300, then tests were also run at $N+100$ and $N-100$.)

Given that N is fixed at a near-optimal value, the experimental design compared the FDCs at this point for all n versions across all example programs subject to test case generation. Assume that the number of program examples is e , and represent the FDC by the letter p_{ij} (i.e., the proportion of decision coverage attained by applying version i to example j ; $i = 1, \dots, n$; $j = 1, \dots, e$).

The data required for the statistical evaluation was arranged as indicated in Table 4.1. An analysis of variance was performed to determine if there were significant differences between the versions and the example programs under test.

4.2 EXPERIMENTAL RESULTS

Because the main portion of the effort of the first six months of Phase 2 has been applied to adapting the working prototype to utilize the new rule structure, no experimental results are available at this point. However, since the prototype is now

Table 4.1. Performance Metrics -- Fractional Decision Coverage

RULE-BASE VERSION	EXAMPLE PROGRAM UNDER TEST			AVERAGE	VARIANCE
	$j=1$	$j=2$		
$i=1$					
$i=2$					
.					
.					
.					
AVERAGE					
VARIANCE					

operational with the new rule base, the test Ada programs have been instrumented, the Librarian is operational, and the experimental design is completed, there are currently no deterrents to initiating the actual test case generation for the example Ada programs by which the data will be generated to evaluate the evolving rule bases. Results are expected in the next few months.

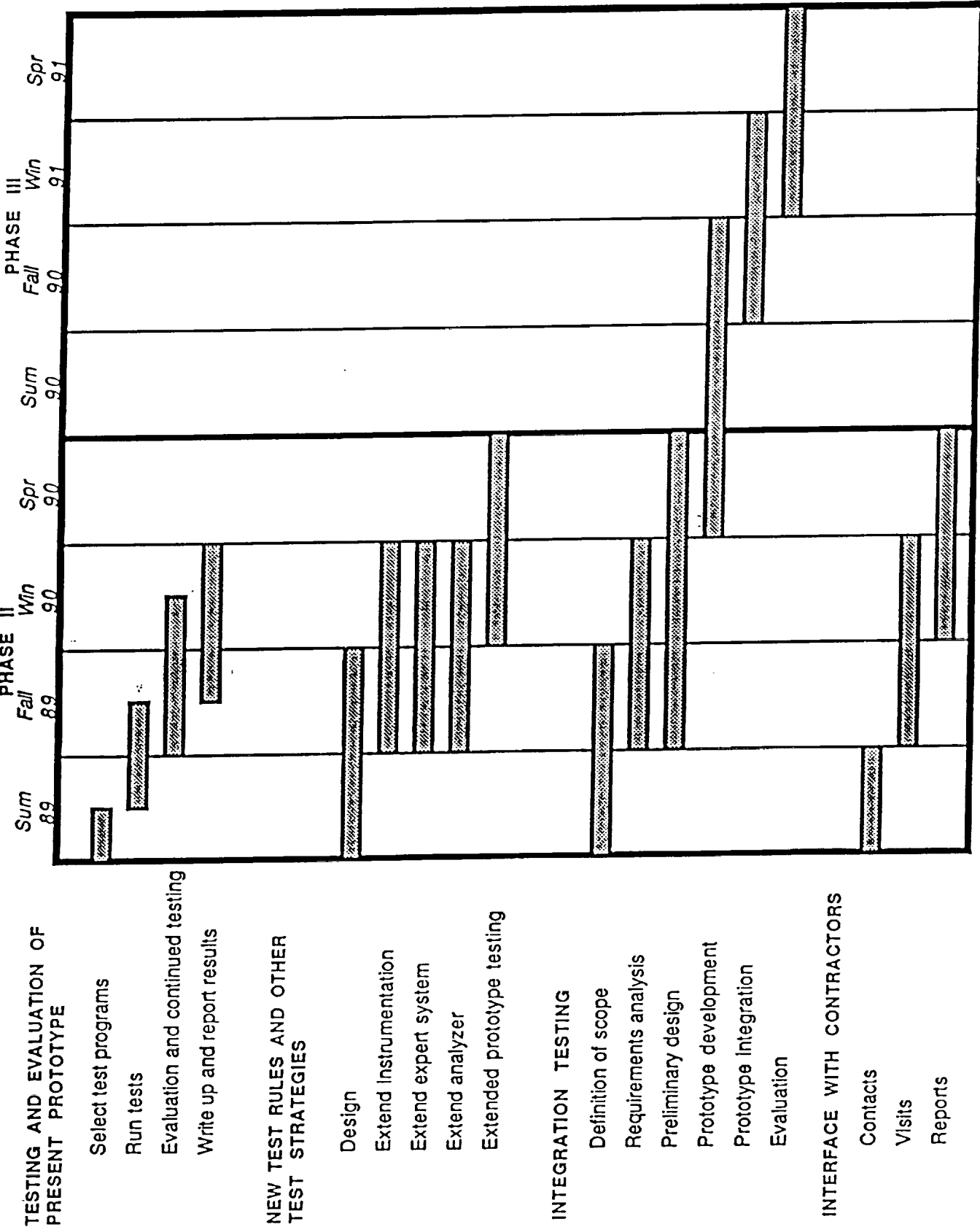
5.0 PROJECT SCHEDULE

The Gantt chart for the project schedule, which was given in the proposal, is presented on the following page. All activities are on schedule, with the exception of the following:

1. Run tests. Test programs have been selected and instrumented but their actual test completion has been deferred until the third quarter of this phase for reasons discussed in Section 4.2. It is expected that this will be initiated immediately and that results will be forthcoming before the end of the winter quarter.
2. Evaluation and continued testing. The end point for this activity should be extended to the middle of spring quarter.
3. Write up and report results. The end point for this activity should be extended to the middle of spring quarter.

All other activities are proceeding on schedule as indicated by the original plan.

TESTING TOOLS FOR ADA SOFTWARE PROPOSED RESEARCH ACTIVITIES



6.0 REFERENCES

- [Brin89] Brindle, A. F., Taylor, R. N., and Jansen, L. R., "A Debugger for Ada Tasking", **IEEE Transactions on Software Engineering**, Vol. 15, No. 3, pp. 293-304, March 1989.
- [Call89] Callahan, D. and Subhlok, J., "Static Analysis of Low-level Synchronization", **Proceedings of the ACM SIGPLAN/-SIGOPS Workshop on Parallel and Distributed Debugging**, published in **ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 100-111, January 1989.
- [Chen87] Cheng, J., Araki, K., and Ushijima, Kazuo, "Event-driven Execution Monitor for Ada Tasking Programs", **Proceedings of COMPSAC 87**, pp. 381-388, October 1987.
- [CLI87] CLIPS Reference Manual, Version 4.1, Artificial Intelligence Section, Johnson Space Center, NASA, September 1987.
- [DEA88] W. H. Deason, "Rule-based Software Test Case Generation," M.S. Thesis, Department of Computer Science and Engineering, December 1988.
- [DEA89] W. H. Deason, D. B. Brown, K. H. Chang, and J. H. Cross II, "A Rule-based Software Test Data Generator," Revised paper submitted to **IEEE Trans. on Knowledge and Data Engineering**, August 1989.
- [DeM78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," **IEEE Computer**, Vol. 11, No. 4, April 1978.
- [Fali82] Falis, Edward, "Design and Implementation in Ada of a Runtime Task Supervisor", **Proceedings of the AdaTEC Conference on Ada**, published in **ACM SIGPLAN Notices**, pp. 1-9, October 1982.
- [Fran88] Franscesco, N. D. and Vaglini, G., "Description of a Tool for Specifying and Prototyping Concurrent Programs", **IEEE Transactions on Software Engineering**, Vol. 14, No. 11, pp. 1554-1564, November 1988.
- [Gait86] Gait, J., "A Probe Effect in Concurrent Programs", **Software -Practice and Experience**, Vol. 16, No. 3, pp. 225-233, March 1986.
- [Germ82] German, S. M., Helmbold, D. P., and Luckham, D. C., "Monitoring for Deadlocks in Ada Tasking", **Proceedings of the AdaTEC Conference on Ada**, published in **ACM SIGPLAN Notices**, pp. 10-27, October 1982.

- [Germ84] German, S. M., "Monitoring for Deadlock and Blocking in Ada Tasking", **IEEE Transactions on Software Engineering**, Vol. SE-10, No. 6, pp. 764-777, November 1987.
- [Gold89] Goldszmidt, G. S., Katz, S., and Yemini, S., "Interactive Blackbox Debugging for Concurrent Languages", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging**, published in **ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 271-283, January 1989.
- [Hail82] Hailpern, B. T., **Verifying Concurrent Processes Using Temporal Logic**, Springer-Verlag, Berlin, 1982.
- [Helm85] Helmbold, D. and Luckham, D., "Debugging Ada Tasking Programs", **IEEE Software**, Vol. 2, No. 2, pp. 47-57, March 1985.
- [Helm85b] Helmbold, D. and Luckham, D. C., "Runtime Detection and Description of Deadness Errors in Ada Tasking", **Ada Letters**, Vol. 4, No. 6, pp. 60-72, 1985.
- [HOW86] W.E. Howden, "A Functional Approach to Program Testing and Analysis," **IEEE Trans. on Software Engineering**, Vol. SE-12, No. 10, October 1986.
- [Hseu89] Hsuesh, W. and Daiser, G. E., "Data Path Debugging: Data-oriented Debugging for a Concurrent Programming Language", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging**, published in **ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 236-247, January 1989.
- [Lamp89] Lamport, L., "A Simple Approach to Specifying Concurrent Systems", **Communications of the ACM**, Vol. 32, No. 1, pp. 32-45, January 1989.
- [LeDou85] LeDoux, C. H. and Parker, D. S., "Saving Traces for Ada Debugging", **Ada in Use, Proceedings of the Ada International Conference**, published in **ACM Ada Letters**, Vol. 5, No. 2, pp. 97-108, September 1985.
- [Lewis84] Lewis, T. G., Spitz, K. R., and McKenney, P. E., "An Interleave Principle for Demonstrating Concurrent Programs", **IEEE Software**, Vol. 1, No. 4, pp. 54-64, October 1984.
- [Mill88] Miller, B. P. and Choi, J.D., "A Mechanism for Efficient Debugging of Parallel Programs", **Proceedings of the SIGPLAN '88 Conference on Programming Language and Implementation**, published in **ACM SIGPLAN Notices**, Vol. 23, No. 7, pp. 135-144, July 1988.

- [Mura89] Murata, T., Shender, B., and Shatz, S. M., "Detection of Ada Static Deadlocks Using Petri Net Invariants", **IEEE Transactions on Software Engineering**, Vol. 15, No. 3, pp. 314-325, March 1989.
- [PRA87] R.E. Prather and P. Myers, Jr., "The Path Prefix Software Testing Strategy," **IEEE Trans. on Software Engineering**, Vol. SE-13, No. 7, July 1987.
- [Ston88] Stone, J. M., "Debugging Concurrent Processes: a Case Study", **Proceedings of the SIGPLAN '88 Conference on Programming Language and Implementation**, published in **ACM SIGPLAN Notices**, Vol. 23, No. 7, pp. 145-153, July 1988.
- [Ston89] Stone, J. M., "A Graphical Representation of Concurrent Processes", **Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging**, published in **ACM SIGPLAN Notices**, Vol. 24, No. 1, pp. 226-235, January 1989.
- [Stran81] Stranstrup, J., "Analysis of Concurrent Algorithms", **Lecture Notes in Computer Science Conpar 81 (Conference on Analyzing Problem Classes and Programming for Parallel Computing)**, pp. 217-230, June 1981.
- [Tai85] Tai, K. C., "On Testing Concurrent Programs", **Proceedings of COMPSAC 85**, pp. 310-317, October 1985.
- [Tai86] Tai, K. C., "A Graphical Notation for Describing Executions of Concurrent Ada Programs", **ACM Ada Letters**, Vol. 6, No. 1, pp. 94-103, January 1986.
- [Tayl78] Taylor, R. N. and Osterweil, L. J., "A Facility for Verification, Testing, and Documentation of Concurrent Process Software", **Proceedings of COMPSAC 78**, pp. 36-41, November 1978.
- [Tayl80] Taylor, R. N. and Osterweil, L. J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", **IEEE Transactions on Software Engineering**, Vol. SE-6, No. 3, pp. 265-277, May 1980.
- [Tayl83a] Taylor, R. N., "Complexity of Analyzing the Synchronization Structure of Concurrent Programs", **Acta Informatica**, Vol. 19, pp. 57-84, April 1983.
- [Tayl83b] Taylor, R. N., "A General-Purpose Algorithm for Analyzing Concurrent Programs", **Communications of the ACM**, Vol. 26, No. 5, pp. 362-376, May 1983.

- [Tayl88] Taylor, R. N. and Young, M., "Combining Static Concurrency Analysis with Symbolic Execution", **IEEE Transactions on Software Engineering**, Vol. 14, No. 10, pp. 1499-1511, October 1988.
- [Utte89] Utter, P. S., and Pancake, C. M., "A Bibliography of Parallel Debuggers", **SIGPLAN Notices**, Vol. 24, No. 11, pp. 29-42, Nov., 1989.

APPENDIX A. PAPER: AUTOMATED UNIT LEVEL TESTING
WITH HEURISTIC RULES, BY CARLISLE, CHANG, CROSS AND KELEHER
SUBMITTED TO THE FIFTH CONFERENCE ON ARTIFICIAL
INTELLIGENCE FOR SPACE APPLICATIONS
HUNTSVILLE, ALABAMA, MAY, 1990

AUTOMATED UNIT-LEVEL TESTING WITH HEURISTIC RULES

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

ABSTRACT

Software testing plays a significant role in the development of complex software systems. Current testing methods generally require significant effort to generate meaningful test cases. The QUEST/Ada¹ system is a prototype system designed using CLIPS [NASA87] to experiment with expert system based test case generation. The prototype is designed to test for condition coverage, and attempts to generate test cases to cover all feasible branches contained in an Ada program. This paper reports on heuristics used by the system and the results of tests of the system using various rule sets. The rule sets used for these tests varied according to the degree of knowledge of the boolean conditions in the program.

INTRODUCTION

There are many approaches to software testing, and most require considerable human interaction at a great cost in man hours. The goal of automating this activity is to provide for more cost effective software testing and to avoid human bias or oversight. One class of automated testing tools, the dynamic analysis tools, is characterized by direct execution of the program under test [DEM87]. A test data generator is a dynamic analysis tool designed to assist the user in achieving goals such as statement coverage, condition coverage, or path testing. The difficulties of test data generation are due to the computation efforts, sometimes wasted, in computing infeasible paths or solving arbitrary path predicates, especially if a predicate contains non-linear terms or function calls. Consequently AI approaches must be utilized to avoid these problems.

QUEST/Ada¹ is a prototype system that is designed to experiment with expert system based test case generation. This system seeks to achieve its goals using heuristic rules to choose and generate new test cases. This paper reports on various rule sets designed to achieve condition coverage of Ada programs with increasing amounts of knowledge about the conditions in the Ada program. Knowledge can vary from little information about the input data (requiring random case generation of the appropriate type of input data), to complete symbolic solutions for variables in the conditions under test.

BACKGROUND

Testing

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

Domain testing

Programs run on finite state machines over finite input sets. Consequently it is theoretically possible to prove a program correct by testing it over its input domain. However in general these domains are too large for this type of testing to be feasible. It is therefore assumed that programs of arbitrary large storage requirements run on machines of arbitrary large size and precision. Unfortunately this assumption leads to results that demonstrate the impossibility of an algorithm to determine correctness of a program. [HOW87]

¹Research and development of the QUEST/Ada system has been supported by the National Aeronautics and Space Administration (NASA). Ada is a trademark of the United States Government, Ada Joint Program Office.

Functional testing

Functional testing is the process of attempting to find discrepancies between the program's output and its requirements specification. [MEY78]. In functional testing [BEI84, HOW86] a program is executed over selected input and the results are compared with expected output. Normally nothing is assumed about the internal structure of the program. Rather, test cases are constructed from knowledge of "what the program is supposed to do", i.e. its "function". This is known as the "black box" approach to testing.

Structural testing

Structural or "white box" testing uses the source code control structure of a program to guide the selection of test data [BEI84]. One metric for the selection process is coverage, which is concerned with the number of structural units exercised by a test case. Examples of this metric are

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

Attempts to develop a practical test generation methodology for branch coverage have suggested approaches ranging from random test generation to full program path predicate solutions. Howden [HOW87] has formalized test generation rules to help programmers test their code. Consequently such rules can be considered "expert knowledge" required for effective and automatic test case generation in an expert system test case generator.

Test case generation

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

Branch coverage is currently regarded as a minimal standard of achievement in structural testing [PRA87]. Thus, the goal of an expert system test case generator is to achieve branch coverage by using heuristic rules with execution feedback to generate test cases sufficient to insure that each branch in a program is invoked at least once. Figure 1 gives a system overview of such a test case generation methodology.

Figure 1

To avoid exponential searches, the analysis may be supported by a search strategy such as that proposed by Prather and Myers [Prat87]. This strategy views a software package as a flowgraph with each condition containing a true and false branch. The goal for test cases is to maximize the number of covered branches as recorded in a branch coverage table. The strategy is to select the first condition in a path from the start for which the condition has not yet been tested in both directions, and to generate (if possible) a test case that will drive this condition in the other direction. The idea behind this strategy is that, since some previous test case has reached the condition, it is already "close" to a test value required to drive an alternate branch of the condition.

AN INTELLIGENT TEST DATA GENERATION SYSTEM

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

Figure 2

Initial test cases are needed to start the process. These may be provided by the user or generated by the system using an initial test case generation rule. Upon execution of the program on test cases, coverage analysis determines what branches have been covered and which branches need further testing. Coverage analysis is basically a table filling process recording the execution of each condition of the program. The expert system generates new test cases by applying rules based on knowledge about both the conditions not yet fully covered, and previous conditions in the execution path that lead to the condition not fully covered. New test cases are generated, and the testing continues. Execution stops when full coverage is indicated, or when a test case limit is reached. Implementation details of the QUEST/Ada system are described in [BRO89].

Rule Based Test Case Generation

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

Initial cases

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

Best test case selection

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

$$\text{ABS}(\text{LHS} - \text{RHS})/2 * \text{MAX}(\text{ABS}(\text{LHS}), \text{ABS}(\text{RHS})).$$

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

affecting the condition under consideration will have a smaller impact on previous conditions when the left hand side and right hand side are far apart.

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

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

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

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

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

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

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

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

Test case generation

In order to experiment with the effects of altering the knowledge about the conditions of a program under test, three categories of rules have been selected. The rules are in the syntax of "CLIPS" [NASA87], a forward chaining expert system tool used by the QUEST/Ada prototype. Comments (lines beginning with ;) are intended to explain the action of the rule. The first category of rule reflects only "type" (integer, float, etc.) knowledge about the variables contained in the conditions. These rules generate new test cases by randomly generating values. The following listing provides an example of this type of rule.

Listing 1.

```
(defrule generate_random_test_cases ""
  (types $?type_list)
;use only type and
  (low_bounds $?low_bounds_list)
;boundary info
  (high_bounds $?high_bounds_list)
;to avoid run error
  =>
;set up a loop to generate n test cases for the
;n input variables
  (bind ?outer_pointer 1)
  (while (<= ?outer_pointer (length $?type_list))
;get test case number
    (bind ?test_number (test_number))
    (format test-case-file "%d" ?test_number)
;step thru each variable
    (bind ?inner_pointer 1)
    (while (<= ?inner_pointer (length $?type_list))
;get the type of the variable
      (bind ?type (nth ?inner_pointer $?type_list))
;assign it a random value
      (bind ?random_value (rand()))
;get range information
```

```

    (bind ?low_bound
      (nth ?inner_pointer $?low_bounds_list))
    (bind ?high_bound
      (nth ?inner_pointer $?high_bounds_list))
;be sure random value is within bounds
    (if (> ?random_value ?high_bound) then
      (bind ?test_value
        (* (/ ?high_bound ?random_value) ?high_bound))
      else
        (bind ?test_value ?random_value))
    (if (< ?random_value ?low_bound) then
      (bind ?test_value
        (* (/ ?low_bound ?random_value) ?low_bound))
      else
        (bind ?test_value ?random_value))
;write value for the variable to the test case file
;in appropriate format
    (if (eq ?type int) then
      (format test-case-file " %d" ?test_value))
    (if (eq ?type fixed) then
      (format test-case-file " %f" ?test_value))
    (if (eq ?type float) then
      (format test-case-file " %e" ?test_value))
;next variable in test case
    (bind ?inner_pointer (+ ?inner_pointer 1)))
    (fprintf test-case-file "\n")
;next test case
    (bind ?outer_pointer (+ ?outer_pointer 1)))
  )

```

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

Listing 2.

```

(defrule generate_increment_by_one_percent_test_cases ""
  (types $?type_list)
  (low_bounds $?low_bounds_list)
  (high_bounds $?high_bounds_list)
;match any condition that is only half covered
  (coverage_table ?decision ?condition true|false)
;get the best test case for each condition
  (best_test_case ?decision ?condition $?values)
=>
  (bind ?outer_pointer 1)
  (while (<= ?outer_pointer (length $?values))
    (bind ?test_number (test_number))
    (format test-case-file " %d" ?test_number)
    (bind ?inner_pointer 1)
    (while (<= ?inner_pointer (length $?values))
      (bind ?type (nth ?inner_pointer $?type_list))

```

```

(bind ?high_bound
  (nth ?inner_pointer $?high_bounds_list))
(bind ?low_bound
  (nth ?inner_pointer $?low_bounds_list))
;increment the current variable by one percent of
;its range
(bind ?one_percent (/ (- ?high_bound ?low_bound) 100))
(bind ?increment
  (+ (nth ?inner_pointer $?values) ?one_percent))
;if this is the variable we want to alter
(if (= ?outer_pointer ?inner_pointer) then
  (if (<= ?increment ?high_bound) then
    (bind ?test_value ?increment)
  else
    (bind ?test_value ?low_bound))
else
  )
;and the other variables are written as is
(bind ?test_value (nth ?inner_pointer $?values)))
(if (eq ?type int) then
  (format test-case-file " %d" ?test_value))
(if (eq ?type fixed) then
  (format test-case-file " %f" ?test_value))
(if (eq ?type float) then
  (format test-case-file " %e" ?test_value))
(bind ?inner_pointer (+ ?inner_pointer 1)))
(fprintout test-case-file crlf)
(bind ?outer_pointer (+ ?outer_pointer 1)))
)

```

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

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

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

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

Listing 3.

```

(defrule generate_symbolic_approximation_plus_increment_test_cases ""

;type information here
(types $?type_list)
(low_bounds $?low_bounds_list)
(high_bounds $?high_bounds_list)
;knowledge about the condition here
(coverage_table ?decision ?condition true|false)
(best_test_case ?decision ?condition $?values)
(value_at_cond ?decision ?condition $?vacs)
(symbolic_boundary ?decision ?condition $?boundaries)
=>
(bind ?outer_pointer 1)
(while (<= ?outer_pointer (length $?values))
  (bind ?test_number (test_number))
  (format test-case-file " %d" ?test_number)
  (bind ?inner_pointer 1)
  (while (<= ?inner_pointer (length $?values))
    (bind ?type (nth ?inner_pointer $?type_list))
;for the variable under consideration
    (if (= ?outer_pointer ?inner_pointer) then
;for its range
      (bind ?high_bound
        (nth ?inner_pointer $?high_bounds_list))
      (bind ?low_bound
        (nth ?inner_pointer $?low_bounds_list))
;get its input value
      (bind ? (nth ?inner_pointer $?values))
;and its value at condition
      (bind ?Xc (nth ?inner_pointer $?vacs))
;and the boundary of the condition
      (bind ?Xb (nth ?inner_pointer $?boundaries))
;generate a guess as to an input value leading to boundary
      (bind ?approximation (* (/ ?Xi ?Xc) Xb))
;generate a small amount to move around boundary
      (if (< (abs ?high_bound) (abs ?low_bound)) then
        (bind ?small_bound ?high_bound)
      else
        (bind ?small_bound ?low_bound))
      (bind ?digit 0)
      (while (!= (trunc ?low_bound) ?low_bound)
        (bind ?digit (+ ?digit 1))
        (bind ?low_bound (* ?low_bound (** 10 ?digit))))
;call it e
      (bind ?e (* 10 (* -1 ?digit)))
      (bind ?incremented_approximation
        (+ ?approximation ?e))
;increment the approximation by e
      (if (<= ?incremented_approximation ?high_bound) then
        (bind ?test_value ?incremented_approximation)
      else
        (bind ?test_value ?high_bound))

```

```

else
  (bind ?test_value (nth ?inner_pointer $?values)))
;write to test case file in appropriate format
  (if (eq ?type int) then
    (format test-case-file "%d" ?test_value))
  (if (eq ?type fixed) then
    (format test-case-file "%f" ?test_value))
  (if (eq ?type float) then
    (format test-case-file "%e" ?test_value))
  (bind ?inner_pointer (+ ?inner_pointer 1)))
(fprintout test-case-file crlf)
;next test case
  (bind ?outer_pointer (+ ?outer_pointer 1)))
)

```

CONCLUSION

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

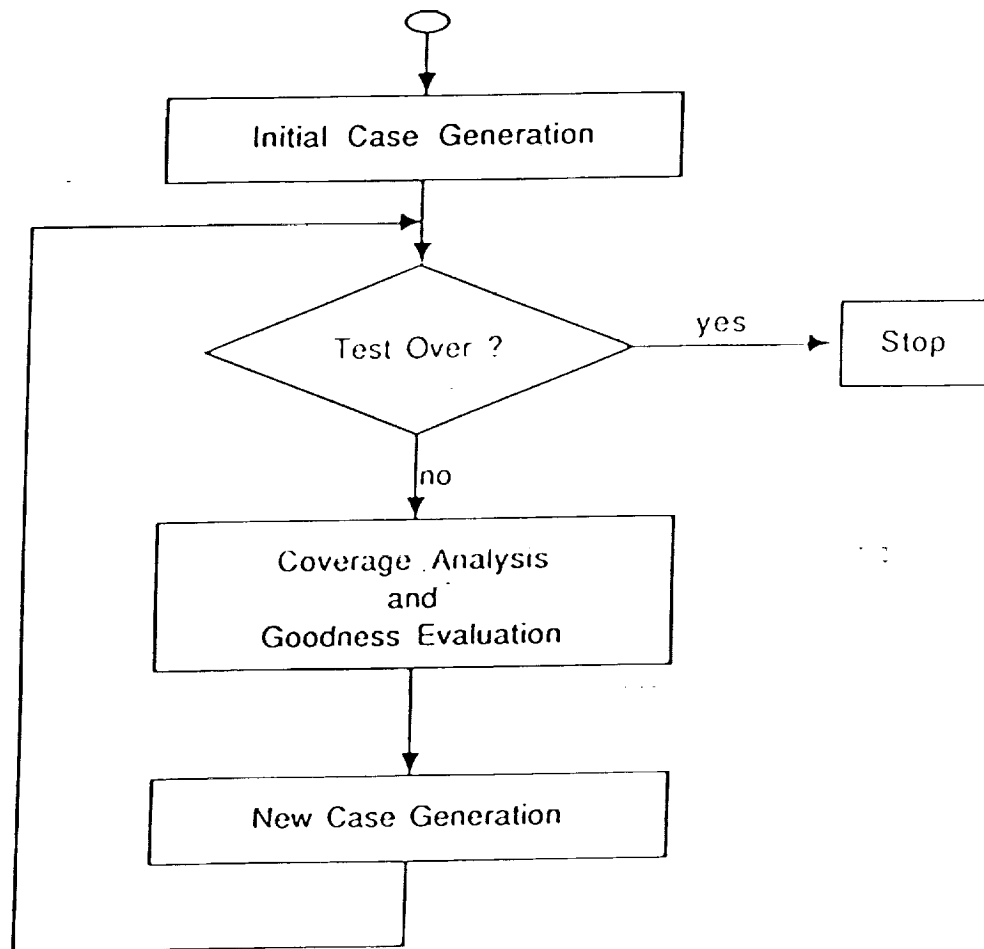
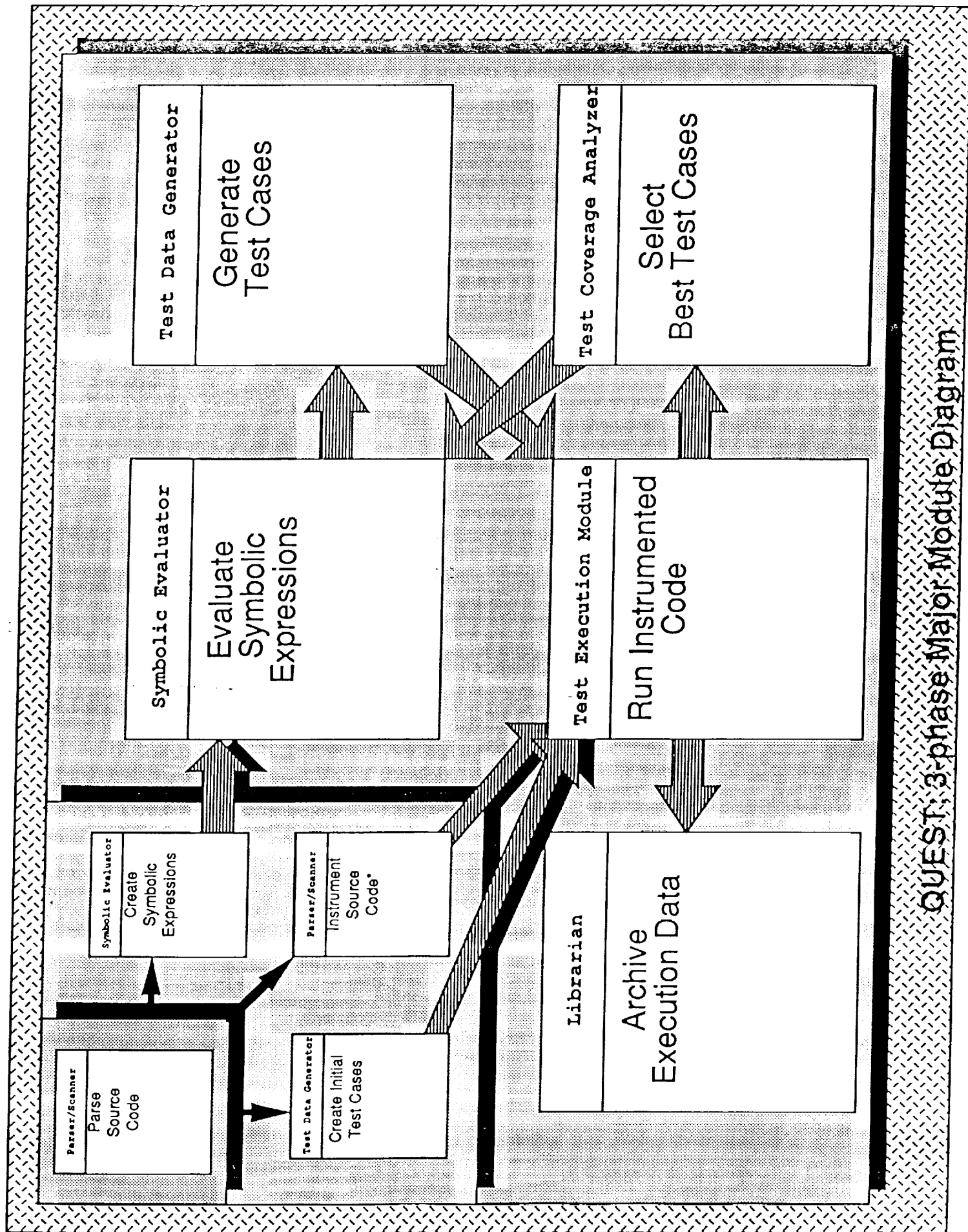


Figure 1 System Concept of the Intelligent Test Data Generator.



QUEST: 3-phase Major Module Diagram

APPENDIX B. EXAMPLE OF INSTRUMENTED PROGRAMS

FTRIANGLE_I

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

```
procedure driver_ftriangle is
  TestNum:          integer;
  indata,outdata:   file_type;
  sidel,side2,side3:FLOAT;
  rval:            integer;
```

```
  procedure print_parms(intermediate: in file_type);
  package inst is new instrumentation(print_parms);
  use inst;
  package inst1 is new inst.float_inst(float);
  use inst1;
  package inst2 is new inst.integer_inst(integer);
  use inst2;
  package int_io is new text_io.integer_io(integer);
  use int_io;
  package float_io is new text_io.float_io(float);
  use float_io;
```

```
  procedure print_parms(intermediate: in file_type) is
  begin
    put(intermediate, sidel);
    put(intermediate, side2);
    put(intermediate, side3);
  end print_parms;
```

```
  function TRIANGLE( SIDE1,SIDE2,SIDE3:in FLOAT ) return INTEGER
  is
```

```
-- returns      0 - not a triangle or SIDE3 not hypotenuse
--              1 - small acute
--              2 - small acute & isosceles
--              3 - small right
--              4 - small obtuse
--              5 - small obtuse & isosceles
--              6 - medium acute
--              7 - medium acute & isosceles
--              8 - medium right
--              9 - medium obtuse
--             10 - medium obtuse & isosceles
--             11 - large acute
--             12 - large acute & isosceles
--             13 - large right
--             14 - large obtuse
--             15 - large obtuse & isosceles
```

```
  RETURN_VAL: INTEGER;
```

```

begin
    if decision(TestNum,1,
        relop(TestNum,1,1,
            ABS(SIDE3*SIDE3-SIDE1*SIDE1+SIDE2*SIDE2),
            LT,0.1))
    then RETURN_VAL := 3;
    elsif decision(TestNum,2,
        relop(TestNum,2,1,
            SIDE1*SIDE1+SIDE2*SIDE2,
            LT,SIDE3*SIDE3))
    then
    if decision(TestNum,3,
        relop(TestNum,3,1,
            SIDE1+SIDE2,
            LT,SIDE3))
    then RETURN_VAL := 0;
    elsif decision(TestNum,4,
        relop(TestNum,4,1,
            ABS(SIDE1-SIDE2),
            LT,0.1))
    then RETURN_VAL := 5;
    else RETURN_VAL := 4;
    end if;
    elsif decision(TestNum,5,
        relop(TestNum,5,1,
            SIDE1,
            GT,SIDE3)
        relop(TestNum,5,2,
            SIDE2,
            GT,SIDE3))
    then RETURN_VAL := 0;
    elsif decision(TestNum,6,
        relop(TestNum,6,1,ABS(SIDE1-SIDE2),LT,0.1))
    then RETURN_VAL := 2;
    else
        RETURN_VAL := 1;
    end if;

    if decision(TestNum,7,
        relop(TestNum,7,1,RETURN_VAL,EQ,0)) then
        return(0);
    elsif
        decision(TestNum,8,relop(TestNum,8,1,SIDE1,GT,10.0)
            and
                relop(TestNum,8,2,SIDE2,GT,10.0))
    then RETURN_VAL := RETURN_VAL + 10;
    elsif
        decision(TestNum,9,relop(TestNum,9,1,SIDE1,GT,1.0)
            and
                relop(TestNum,9,2,SIDE2,GT,1.0))
    then RETURN_VAL := RETURN_VAL + 5;

```

```

        end if;

        return(RETURN_VAL);
    end;

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

    while not End_OF_file(indata) loop
        get(indata,TestNum);          --TestNum,param1,param2,...
        get(indata,side1);
        get(indata,side2);
        get(indata,side3);

        rval := triangle(side1,side2,side3);

        put(outdata,TestNum);
        --TestNum,modifiable1,modifiable2,...
        put(outdata,rval);
        new_line(outdata);
    end loop;

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

```

```

ITRIANGLE_I

with text_io, instrumentation;
use text_io;

procedure driver_itriangle is
  TestNum:      integer;
  indata,
  outdata:      file_type;

  side1,side2,side3,rval:  integer;

  procedure print_parms(intermediate: in file_type);
  package inst is new instrumentation(print_parms);
  package inst1 is new inst.integer_inst(integer);
  use inst,inst1;
  package int_io is new text_io.integer_io(integer);
  use int_io;

  procedure print_parms(intermediate: in file_type) is
  begin
    put(intermediate, side1);
    put(intermediate, side2);
    put(intermediate, side3);
  end print_parms;

  function ITRIANGLE( side1,side2,side3:in INTEGER ) return
  INTEGER is

    return_val: INTEGER;

    -- returns      0 - not a triangle or side3 not hypotenuse
    --              1 - small acute
    --              2 - small acute & isosceles
    --              3 - small right
    --              4 - small obtuse
    --              5 - small obtuse & isosceles
    --              6 - medium acute
    --              7 - medium acute & isosceles
    --              8 - medium right
    --              9 - medium obtuse
    --              10 - medium obtuse & isosceles
    --              11 - large acute
    --              12 - large acute & isosceles
    --              13 - large right
    --              14 - large obtuse
    --              15 - large obtuse & isosceles

  begin
    if decision(TestNum,1,

```

```

        relop(TestNum,1,1,side3*side3,EQ,side1*side1+
            side2*side2)) then
            return_val := 3;
        elsif decision(TestNum,2,
            relop(TestNum,2,1,side1*side1+side2*side2,LT,
                side3*side3)) then
            if
decision(TestNum,3,relp(TestNum,3,1,side1+side2,LT,side3)) then
                return_val := 0;
            elsif decision(TestNum,4,relp(TestNum,4,1,side1,EQ,side2))
                then
                    return_val := 5;
            else
                return_val := 4;
            end if;
        elsif decision(TestNum,5,relp(TestNum,5,1,side1,GT,side3)
            or relp(TestNum,5,2,side2,GT,side3)) then
            return_val := 0;
        elsif decision(TestNum,6,relp(TestNum,6,1,side1,EQ,side2))
            then
                return_val := 2;
            else
                return_val := 1;
            end if;

        if decision(TestNum,7,relp(TestNum,7,1,return_val,EQ,0))
            then
                return(0);
        elsif decision(TestNum,8,relp(TestNum,8,1,side1,GT,10)
            and relp(TestNum,8,2,side2,GT,10)) then
            return_val := return_val + 10;
        elsif decision(TestNum,9,relp(TestNum,9,1,side1,GT,1) and
            relp(TestNum,9,2,side2,GT,1)) then
            return_val := return_val + 5;
        end if;

        return(return_val);

    end;

```

```

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

    while not End_OF_file(indata) loop
        get(indata,TestNum);           --TestNum,param1,param2,...
        get(indata,side1);
        get(indata,side2);
    end loop;

```

```

get(indata,side3);

rval := itriangle(side1,side2,side3);

put(outdata,TestNum);
--TestNum,modifiable1,modifiable2,...
put(outdata,rval);
new_line(outdata);
end loop;

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

```


MAX3_I

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

```
procedure driver_max3 is  
  TestNum:      integer;  
  indata,      integer;  
  outdata:      file_type;  
  i,j,k,rval:   integer;
```

```
  procedure print_parms(intermediate: in file_type);  
  package inst is new instrumentation(print_parms);  
  package inst1 is new inst.integer_inst(integer);  
  use inst,inst1;  
  package int_io is new text_io.integer_io(integer);  
  use int_io;
```

```
  procedure print_parms(intermediate: in file_type) is  
  begin  
    put(intermediate, i);  
    put(intermediate, j);  
    put(intermediate, k);  
  end print_parms;
```

```
  function MAX3(I, J, K: in INTEGER) return INTEGER is  
    L: INTEGER;  
  begin  
    -- compute the maximum of I and J  
    if decision(TestNum,1,relap(TestNum,1,1,I,GT,J)) then  
      L := I;  
    else  
      L := J;  
    end if;  
  
    -- compute the maximum of I, J, and L  
    if decision(TestNum,2,relap(TestNum,2,1,L,LT,K)) then  
      L := K;  
    end if;  
  
    return(L);  
  end;
```

```
begin  
  open(indata,in_file,"test.data");  
  create(intermediate,out_file,"intermediate.results");  
  create(outdata,out_file,"output.data");  
  
  while not End_OF_file(indata) loop
```

```

        get(indata,TestNum);          --TestNum,parm1,parm2,...
        get(indata,i);
        get(indata,j);
        get(indata,k);

        rval := max3(i,j,k);

        put(outdata,TestNum);
--TestNum,modifiable1,modifiable2,...
        put(outdata,rval);
        new_line(outdata);
    end loop;

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

TEST1_I

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

```
procedure driver_test1 is  
  TestNum:      integer;  
  indata,      outdata:      file_type;
```

```
  i,j,k:      integer;
```

```
  procedure print_parms(intermediate: in file_type);  
  package inst is new instrumentation(print_parms);  
  use inst;  
  package inst1 is new inst.integer_inst(integer);  
  use inst1;  
  package int_io is new text_io.integer_io(integer);  
  use int_io;
```

```
  procedure print_parms(intermediate: in file_type) is  
  begin  
    put(intermediate, i);  
    put(intermediate, j);  
    put(intermediate, k);  
  end print_parms;
```

```
  procedure test1(i: in out integer;  
                  j: in out integer;  
                  k: in out integer) is  
  begin  
    while decision(TestNum,1,  
                  relop(TestNum,1,1,i,GT,j)) loop    --d1  
      i := i - 1;  
      k := (k + 314) mod 25;  
      if decision(TestNum,2,  
                  relop(TestNum,2,1,i,GT,k)) then --d2  
        while decision(TestNum,3,  
                        relop(TestNum,3,1,i,GT,k)) loop --d3  
          k := k + 1;  
          if decision(TestNum,4,  
                      relop(TestNum,4,1,k,GE,27)) then --d4  
            null;  
          else  
            null;  
          end if;  
        end loop;  
      else  
        if decision(TestNum,5,  
                    relop(TestNum,5,1,i,LT,k-3)) then --d5  
          if decision(TestNum,6,
```

```

                                relop(TestNum,6,1,i-10,LT,j))
                                then  --d6
                                    null;
                                else
                                    null;
                                end if;
                            else
                                while decision(TestNum,7,
                                    relop(TestNum,7,1,i,GE,k-3))
                                    loop  --d7
                                        i := i - 1;
                                    end loop;
                                end if;
                            end if;
                        end loop;
                        if decision(TestNum,8, relop(TestNum,8,1,i,EQ,j)) then
                            --d8
                                null;
                        else
                            null;
                        end if;
                    end test1;

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

    while not End_OF_file(indata) loop
        get(indata,TestNum);          --TestNum,parml,parm2,...
        get(indata,i);
        get(indata,j);
        get(indata,k);

        test1(i,j,k);

        put(outdata,TestNum);
        --TestNum,modifiable1,modifiable2,...
        put(outdata,i);
        put(outdata,j);
        put(outdata,k);
        new_line(outdata);
    end loop;

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

```

TEST2_I

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

```
procedure driver_test2 is  
  TestNum:      integer;  
  indata,        
  outdata:      file_type;  
  a,b:          integer;
```

```
  procedure print_parms(intermediate: in file_type);  
  package inst is new instrumentation(print_parms);  
  use inst;  
  package inst1 is new inst.integer_inst(integer);  
  use inst1;  
  package int_io is new text_io.integer_io(integer);  
  use int_io;
```

```
  procedure print_parms(intermediate: in file_type) is  
  begin  
    put(intermediate, a);  
    put(intermediate, b);  
  end print_parms;
```

```
procedure test2(a: in out integer; b: in out integer) is  
  c,d: integer;  
  begin  
    d := 2;  
    while decision(TestNum,1,relon(TestNum,1,1,a,LT,1)) loop  
      if decision(TestNum,2,relon(TestNum,2,1,a,GT,b)) then  
        c := 713 mod a;  
        while decision(TestNum,3,  
          relon(TestNum,3,1,c,GT,a)) loop  
          c := c - 2;  
          d := d - 1;  
          if decision(TestNum,4,  
            relon(TestNum,4,1,c,GT,d)) then  
            d := d-2;  
          else  
            null;  
          end if;  
          if decision(TestNum,5,  
            relon(TestNum,5,1,c,LT,b)) then  
            if decision(TestNum,6,  
              relon(TestNum,6,1,c,LT,213 mod b)) then  
              if decision(TestNum,7,  
                relon(TestNum,7,1,b,GT,d)) then  
                null;  
              else  
                if decision(TestNum,8,
```

```

                                relop(TestNum,8,1,b,EQ,d)) then
                                    b := b+1;
                                else
                                    null;
                                end if;
                            end if;
                        else
                            c := 213 mod b;
                            end if;
                        else
                            null;
                            end if;
                    end loop;
                else
                    if decision(TestNum,9,relon(TestNum,9,1,a,EQ,b)) then
                        a := b-5;
                        while decision(TestNum,10,
                                relop(TestNum,10,1,a,GT,b)) loop
                            a := a-1;
                            b := (b*b*a*a) mod 13;
                        end loop;
                    else
                        if decision(TestNum,11,
                                relop(TestNum,11,1,a,LT,b)) then
                            a := a+1;
                        else
                            null;
                        end if;
                    end if;
                end loop;
            end test2;
        begin
            open(indata,in_file,"test.data");
            create(intermediate,out_file,"intermediate.results");
            create(outdata,out_file,"output.data");

            while not End_OF_file(indata) loop
                get(indata,TestNum);
                get(indata,a);
                get(indata,b);

                test2(a,b);

                put(outdata,TestNum);
                put(outdata,a);
                put(outdata,b);
                new_line(outdata);
            end loop;

```

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

TEST3_I

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

```
procedure driver_test3 is
  TestNum:      integer;
  indata,
  outdata:      file_type;
  i,j:          integer;
```

```
  procedure print_parms(intermediate: in file_type);
  package inst is new instrumentation(print_parms);
  use inst;
  package inst1 is new inst.integer_inst(integer);
  use inst1;
  package int_io is new text_io.integer_io(integer);
  use int_io;
```

```
  procedure print_parms(intermediate: in file_type) is
  begin
    put(intermediate, i);
    put(intermediate, j);
  end print_parms;
```

```
procedure test3(i,j:in out integer) is
```

```
  k: integer;
  begin
```

```
    k := 0;
```

```
    while decision(TestNum,1,relap(TestNum,1,1,j,LT,50)) loop
```

```
      if decision(TestNum,2,relap(TestNum,2,1,i,EQ,j)) then
```

```
        i := i+1;
```

```
        j := j-1;
```

```
        k := j+1;
```

```
      else
```

```
        j := j+1;
```

```
        k := i;
```

```
      end if;
```

```
    end loop;
```

```
    while decision(TestNum,3,relap(TestNum,3,1,i,LE,k-3)) loop
```

```
      i := i+3;
```

```
    end loop;
```

```
    if decision(TestNum,4,relap(TestNum,4,1,i,EQ,j)) then
```

```
      null;
```

```
    else
```

```
      if decision(TestNum,5,relap(TestNum,5,1,i,EQ,k)) then
```

```
        null;
```

```
      end if;
```



```

        end if;
    end test3;

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

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

        test3(i,j);

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

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

```

LINEAR_I

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

```
procedure driver_linear is
```

```
    TestNum:      integer;
    indata,outdata: file_type;
    y,z,rval:     integer;
    x:            float;
```

```
    procedure print_parms(intermediate: in file_type);
    package inst is new instrumentation(print_parms);
    use inst;
    package inst1 is new inst.integer_inst(integer);
    use inst1;
    package inst2 is new inst.float_inst(float);
    use inst2;
    package int_io is new text_io.integer_io(integer);
    use int_io;
    package float_io is new text_io.float_io(float);
    use float_io;
```

```
    procedure print_parms(intermediate: in file_type) is
    begin
```

```
        put(intermediate, x);
        put(intermediate, y);
        put(intermediate, z);
    end print_parms;
```

```
function LINEAR( X:in FLOAT;Y,Z: in INTEGER ) return INTEGER
is
```

```
begin
```

```
    if decision(TestNum,1,relon(TestNum,1,1,X,GT,10.5)) then
        if decision(TestNum,2,relon(TestNum,2,1,Y,EQ,2) and
            relon(TestNum,2,2,Z,EQ,52)) then
            if decision(TestNum,3,
                relon(TestNum,3,1,X,GT,FLOAT(2*Y+15))) then
                return(1);
            elsif decision(TestNum,4,
                relon(TestNum,4,1,X,GT,FLOAT(-2*Y+15))) then
                return(2);
            end if;
        elsif decision(TestNum,5,relon(TestNum,5,1,Y,GT,2) and
            relon(TestNum,5,2,Z,GT,52)) then
            if decision(TestNum,6,
                relon(TestNum,6,1,X,GT,19.2)) then
                return(3);
            else
```

```

        return(4);
    end if;
end if;
elseif decision(TestNum,7,relon(TestNum,7,1,X,LT,10.0) and
                relon(TestNum,7,2,Y,GT,10*Z)) then
    if decision(TestNum,8,relon(TestNum,8,1,Y,EQ,100)) then
        return(5);
    else
        return(6);
    end if;
else
    return(7);
end if;
end;

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

    while not End_OF_file(indata) loop
        get(indata,TestNum);          --TestNum,param1,param2,...
        get(indata,x);
        get(indata,y);
        get(indata,z);

        rval := linear(x,y,z);

        put(outdata,TestNum);
--TestNum,modifiable1,modifiable2,...
        put(outdata,rval);
        new_line(outdata);
    end loop;

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

```

APPENDIX C. LIBRARIAN ROUTINES

The librarian routines can be divided into three main parts: archive association, archive data set manipulation, and QUEST/Ada specific routines.

The archive association routines are:

```
lib_init()
lib_end()
lib_set()
lib_directory()
lib_remove()
```

The data set manipulation routines are:

```
lib_open()
lib_close()
lib_read()
lib_write()
lib_update()
lib_set_key()
lib_key_pattern()
```

The QUEST/Ada specific routines are:

```
lib_quest_setup()
lib_quest_connect()
lib_quest_shutdown()
lib_archive_results()
```

The QUEST/Ada routines are all that need to be called by other components of the QUEST/Ada system (such as the test generation module). Each of the above routines will be documented below in terms of function, arguments and return values.

```
int lib_init( lib_database)
           db_definition *lib_database;
```

Description:

The function `lib_init` initializes the librarian's data structures. No archive is associated with the initialization. Function `lib_init` needs only to be called once during a program's execution and must be called before any other librarian routine.

Argument:

`lib_database` is a pointer to a database definition type. This is for future expansion. Currently, passing NULL is sufficient for setting up the librarian for QUEST/Ada data set manipulation.

Return Value:

Librarian result code.

```
int lib_end( lib_database)
    db_definition *lib_database;
```

Description:

The function lib_end allows the librarian to clean up before termination. The librarian will have to be initialized again before it can be used after a call to lib_end.

Argument:

lib_database is a pointer to a database definition type. This is for future expansion (allowing multiple databases to be active). Passing NULL is sufficient for the QUEST/Ada implementation.

Return Value:

Librarian result code.

```
int lib_set( arch_name, options)
    char      *arch_name;
    unsigned  options;
```

Description:

The function lib_set associates the librarian with a specific archive. If the appropriate option is set, the archive will be created if it does not exist. An archive must be accessed via lib_set before any of its data sets can be manipulated.

Arguments:

arch_name is a character string representing the name of the archive system. This is not a file name, and it should not include any directory information (see lib_directory).

options is an unsigned integer consisting of a number of flags set to represent options in handling the archive (defined in file librarian.h):

LIB_CREATE	- Create if not present.
LIB_READ	- Reads are allowed.
LIB_WRITE	- Writes are allowed.
LIB_UPDATE	- Updates are allowed.
LIB_DELETE	- Deletes are allowed.
LIB_GEN_ACCESS	- All above options turned on.

Note that in most cases an archive will be opened with option set to LIB_GEN_ACCESS so that all actions are valid.

Return Value:

Librarian result code.

```
int lib_directory( directory)
    char *directory;
```

Description:

The function `lib_directory` allows the librarian to associate the librarian with a given directory path name. The directory path name should not contain any file name specifications.

Argument:

`directory` is a character string containing an accessible directory path name.

Return Value:

Librarian result code.

```
int lib_remove( arch_name, options)
               char      *arch_name;
               unsigned  options;
```

Description:

The function `lib_remove` deletes all data sets of an archive. The functions `lib_directory` and `lib_set` must usually be called before `lib_remove` can find the data set files.

Arguments:

`arch_name` is the name of the archive system to be removed. It does not contain any directory information.

`options` is a field for future expansion. Currently, passing NULL will be sufficient for a successful call.

Return Value:

Librarian result code.

```
int lib_open( data_set, options)
             unsigned  data_set;
             unsigned  options;
```

Description:

The function `lib_open` attempts to open a data set in an active archive. A data set must be open before being manipulated. Note that if the data set is already opened, it will not be reopened; rather, a count for the data set will be incremented. The data set will not be closed until this count has reached zero. All index files and the data file are opened for the data set.

Arguments:

`data_set` is an unsigned number representing a data set. Data sets start at zero and increment upwards without any gaps. There is a maximum number of data sets that an archive can have.

`options` is an unsigned number representing the operations that are valid for this data set open. It is currently not used and passing NULL will be sufficient.

Return Value:
Librarian result code.

```
int lib_close( data_set)
               unsigned data_set;
```

Description:

The function lib_close decrements the open count for a data set (if it is opened in the first place). If the count reaches zero, then all the index files and the data file are closed.

Argument:

data_set is the number for the data set that is to be closed. Note that data sets start at zero and increment upwards.

Return Value:
Librarian result code.

```
int lib_read( data_set, record, method)
              unsigned data_set;
              void      *record;
              unsigned method;
```

Description:

The function lib_read attempts to locate and read a record existing within an open data set into a given buffer. The record can be located in a variety of ways (governed by the method argument). Note that if this read operation is searching based on keys, then this key should be established by a lib_set_key call before the lib_read call.

For sequential reading, the methods LIB_FIRST_REC and LIB_NEXT_REC should be used. For keyed reading, the methods LIB_FIRST_MATCH and LIB_NEXT_MATCH are available. Note that LIB_NEXT_MATCH is a valid method only if the data set allows for duplicate keys.

Arguments:

data_set is the number of an opened data set for the active archive.

record is the buffer into which the record will be read into (if found).

method is the search method for finding the record:

LIB_FIRST_REC - First record in the data set.

LIB_NEXT_REC - Next record to be read in.

LIB_FIRST_MATCH - First keyed match.

LIB_NEXT_MATCH - Next keyed match.

Return Value:

Librarian result code (note LIB_EOF and LIB_NO_MATCH are not errors).

```

int  lib_write( data_set, record)
            unsigned data_set;
            void      *record;

```

Description:

The function lib_write saves the contents of an open data set's record into the archive. The index files are updated to note the location of the new record in the data file. It is very important that all keys associated with the data set record are established (via lib_set_key) before the call to lib_write, since all index files will be updated.

Arguments:

data_set in the unsigned number representing which data set is to be updated.

record is a pointer to the buffer to be written out. The librarian already knows how many bytes to write out (because of the lib_set call) and the contents of the keys (because of preceding calls to lib_set_key).

Return Value:

Librarian result code (note that lib_write could fail if a duplicate key exists for a key notated to being unique).

```

int  lib_update( data_set, record)
            unsigned data_set;
            void      *record;

```

Description:

The function lib_update replaces the data file's contents for the given record. Note that lib_update does not update the keyed structure for the record, only the data file contents. If the keys need to be changed, lib_delete should be called for the record followed by a lib_write for the new keyed contents.

Arguments:

data_set is an unsigned number reflecting which data set's last record read is to be modified.

record is a pointer to the new data contents of the record being updated.

Return Value:

Librarian result code.

```

int  lib_set_key( data_set, key_number, vargs)
            unsigned data_set;
            unsigned key_number;
            va_list  *vargs;

```


Description:

The function `lib_set_key` is used to establish the contents of a key associated with a data set's record. It must be called before any keyed read and before any write. For reading, only the key that is being used to access the data set needs to be established (the last established key will, in fact, be used as the index into the data file). For writing, all keys for a record must be set before the record is written out.

Arguments:

`data_set` is an unsigned number representing which data set's record is having its key set.

`key_number` is an unsigned number (starting at zero) representing which key is being set for the record.

`vargs` is the actual components of the key. A key can have a number of components, the combination of which are represented by an ASCII null terminated string. A format string for the key (which is identical to a standard `printf` style format string) is established by the archive's `lib_key_pattern` calls. The `vargs` passed to `lib_set_key` are expected to follow the format string. The `vargs` argument is actually passed to a `vsprintf` call.

Return Value:

`LIB_NO_ERROR.`

```
int lib_key_pattern( data_set, key_number, key_pattern)
                    unsigned data_set;
                    unsigned key_number;
                    char      *key_pattern;
```

Description:

The function `lib_key_pattern` should be called after an archive is connected to. It has to be called before any keyed operations can proceed. `lib_key_pattern` establishes a `printf` style format string for the keys of each data set. All keys for a data set are stored in the data set's index files in ASCII string format.

Arguments:

`data_set` is a number indicating which archive data set this key pattern is being set for.

`key_number` is the key for the record whose pattern is be established.

`key_patter` is a `printf` style format string that will later be used in calls to `lib_set_key`. For instance, if the key pattern is `"%d/%d"`, then it is expected that the key will be set with two integers.

Return Value:

`LIB_NO_ERROR.`

```

int  lib_quest_setup( *dir, *name)
    char *dir;
    char *name;

```

Description:

The function lib_quest_setup is a general purpose routine to connect the program to a QUEST/Ada style archive. If a matching archive already exists (same name and in the same directory), it is DELETED. Thus, lib_quest_setup should be used when desiring to output to a new archive and not when adding to an existing one, since the previous version will be deleted. All setup functions are handled and the program can continue with lib_opens and lib_closes.

Arguments:

dir is a character string representing the directory the archive is to be stored under.

name is the system name for the archive. Note that this is not a file name and should not contain any directory information.

Return Value:

Librarian result code.

```

int  lib_quest_connect( *dir, *name)
    char *dir;
    char *name;

```

Description:

The function lib_quest_connect is used to "connect" to an existing archive. Thus, the program is more than likely intending to report on the contents of an existing archive or add to the archive. Function lib_quest_connect handles are setup functions for a QUEST/Ada archive.

Arguments:

dir is a character string representing the directory in which the archive will reside.

name is the system name for the archive. Note that this is not a file name and should not contain any directory information.

Return Value:

Librarian result code.

```

int  lib_quest_shutdown()

```

Description:

This function shuts down an active QUEST/Ada style archive.

Arguments:

None.

Return Value:

Librarian result code.

```
int  lib_archive_results(      generation, list, intermediate_name,
                               testdat_name, testres_name)
int      generation;
struct ir_record_type  *list;
char      *intermediate_name;
char      *testdat_name;
char      *testres_name;
```

Description:

The function lib_archive_results is a general purpose routine that collects all information generated from one QUEST/Ada packet loop and stores into the current archive.

Arguments:

generation is the packet number for the test data.

list is the head node pointer to the coverage table linked list. Pass NULL if this information should not be archived.

intermediate_name is the full path name of the intermediate data file. Pass NULL if this information is not intended to be archived.

testdat_name is the full path name of the test data file. Pass NULL if this information is not to be archived.

testres_name is the full path name of the test results file. Pass NULL if this file is not be archived.