QUEST/Ada

QUERY UTILITY ENVIRONMENT FOR SOFTWARE TESTING OF ADA

The Development of a
Program Analysis Environment
for Ada

Contract Number NASA-NCC8-14

Task 1, Phase 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

August 1990

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

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

FACULTY INVESTIGATORS

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

GRADUATE RESEARCH ASSISTANTS

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

UNDERGRADUATE RESEARCH ASSISTANT

Ken Kirksey

1.0 EXECUTIVE SUMMARY

This report presents the results of research and development efforts of 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 were the following:

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 progress made in considering concurency testing. In Section 5 the results of the evaluations which were performed in the second half of this phase are discussed. These results demonstrate the validity of the rule-based approach toward test case generation in that a comparison between each successive set of rules was performed as they evolved. Section 6 presents a review of the project schedule and the anticipated results from Phase 3 of the project. The appendices present supplementary programs as well as the text of several papers that have been published or submitted for publication as a result of this research.

2.0 LITERATURE REVIEW: CONCURRENCY TESTING

2.1 OVERVIEW OF THE LITERATURE REVIEW

This chapter of the report summaries the literature review which concentrated on concurrency testing. It makes frequent reference to the bibliography of collected papers, which is contained in Section 7. 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. Note that a more general literature review on software testing is given in the Phase I report.

2.2 SIGNIFICANT ARTICLE LIST

2.2.1 STATIC ANALYSIS

Generally, static analysis leaves much to be desired. This is particularly true when concurrency is involved. Static analysis has highly restrictive rules stemming from its inability to deal with dynamic tasks or subscripted references to tasks. Also, it requires the consideration of an extremely large sample space (this is especially seen in 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 criticizes 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 an extension of [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 finding test cases 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 presented 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 (parcelling), 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 considering the literature on the test of code involving concurrency, issues arise in addition to those 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 or 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:

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

<u>Program Scope Information</u>: 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 p_i can invoke the unit p_j. 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.
$\{G_1, ..., G_U\}$	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 $(c_1, c_2, ..., c_{T'})$, where each c_i is either a state node of a flowgraph G_j or 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 is generally more than one). There can be a successor if:

(1) For all i, 1 < = i < = T', either:
  (a) c'_i in succ(c_i),
  (b) $c'_i = c_{i,}$
  (c) c_i = inactive and c'_i = begin task, or
  (d) c_i = end task and c'_i = inactive, and

(2) There exists at least one c'_j, 1 < = j < = T', which represents application of case (a), (b), or (d) above (thus requiring forward movement).

Given the definition for a currency 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 task 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, ..., c_k of concurrency states such that:

(1) c_1 = (begin < <MAIN> >, inactive, ..., inactive), and
(2) For all i, 1 < = i < = k, c_{i+1} in 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 that 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 know 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, Franscesco [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 be deficient. 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 suggests 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 is used to prune the information static analysis generates. 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.

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.

Helmbold [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.

Helmbold 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, Helmbold goes over possible extensions to this method. One calls for keeping track of more information (e.g., 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.

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

He states that "The life cycle of a task can be described by a sequence of states of the task from 1-Staring 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 that of 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 frames (subprograms, blocks, or other tasks) upon which a task depends. Upon termination, the node is removed from the tree.
(2) Entry-call-queue-manager: every time an entry call is made on a task, this component inserts an element into a list which indicates the module which 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 <T1,T2> 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 locally in the task's body. He indicated that there is no good way to generate a task name and 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). These 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 task t2),
(3) removing an edge (task t2 completes 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 from 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 inactive tasks can be identified 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 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, the amount of tasking 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 this 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.

In summary, this literature review forms the basis for the design of components within QUEST which will consider concurrency within the Ada programs which it tests. This design is given in Section 4 below.

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 will be given below). Instrumentation also involves surrounding the test module with a "driver" or "harness". This harness is responsible for passing the test data generated by the rule base to the module under test, either as parameters or global information.

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

The symbolic evaluation rules utilize the coverage table and the symbolic boundary information. The work of the symbolic evaluator is divided into two parts -- developing and evaluating symbolic expressions. Using descriptions of the conditions in the module under test provided by the PSM, the SE develops symbolic boundary expressions in which each of the variables in a condition is represented in terms of the other variables. This boundary expression implicitly describes the point at which the input variables will cause the Boolean condition to evaluate to equivalence. Thus, by adding or subtracting a small value, epsilon, to the boundary, the 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 $D_i(t+1) = D_b(t)*(D_i(t)/D_c(t))$.

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

Finally, a data management facility 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., (upper limit - 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 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 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.

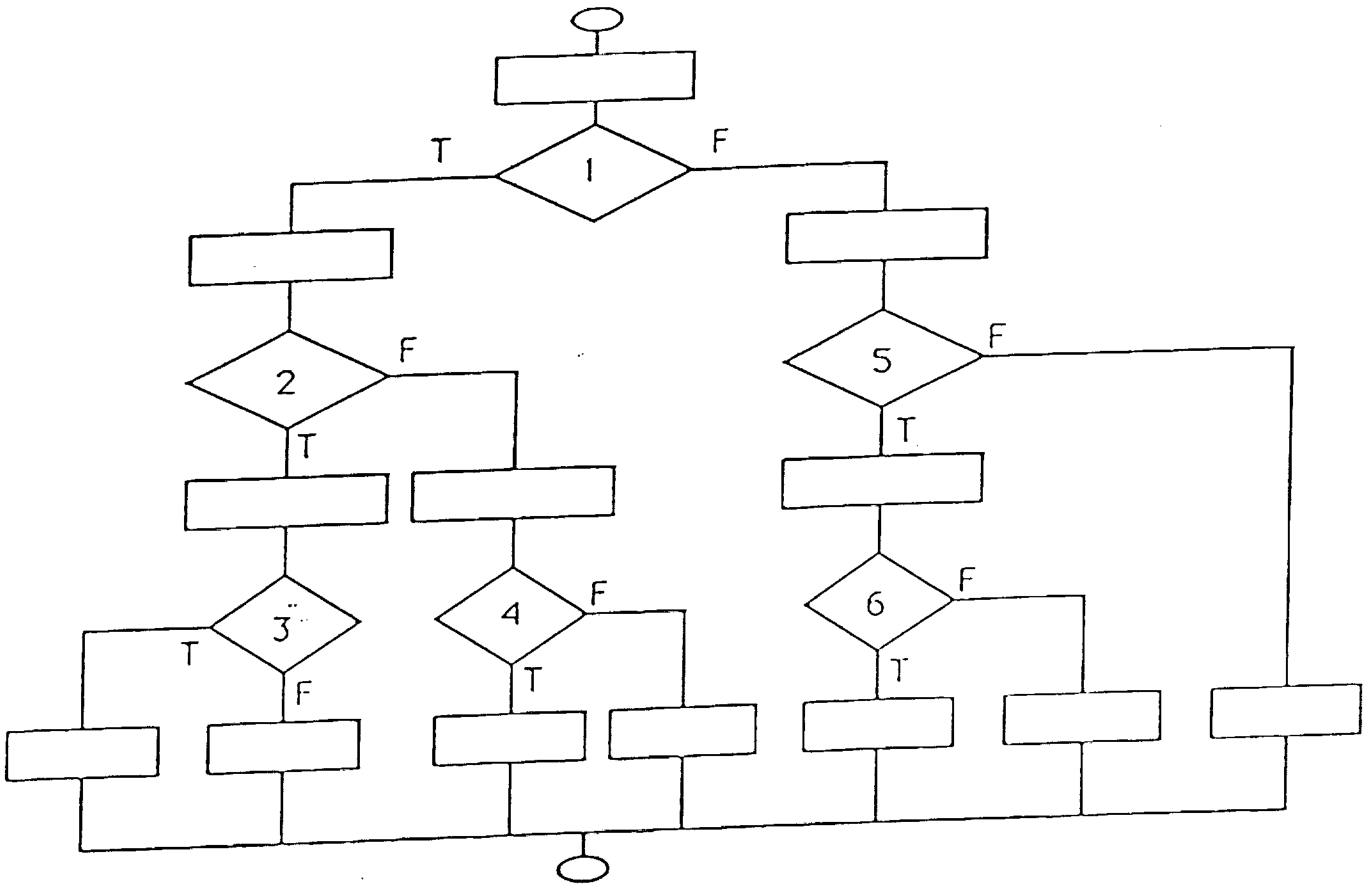


Figure 3.2a A sample flow chart

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

Figure 3.2b Coverage table of Figure 3.2a

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

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

Lhs(t1) and rhs (t1) represent the evaluated value of lhs and rhs, respectively, when t1 is used as the input data. This measure tells the closeness between lhs and rhs [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

$$|\text{lhs (t1)} - \text{rhs (t1)}| \,/\, (2{*}\text{MAX}\,(|\text{lhs (t1)}|, |\text{rhs (t1)}|)) \qquad (2)$$

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 slightness of modification of a test case as its goodness, this measure is computed based on the target condition only. A slight modification to the lhs and rhs of the target condition may not have the same meaning to those conditions on the path. This may result in unanticipated branchings along the path, therefore losing the original purpose of the new cases. In order to reduce the likelihood of unanticipated branching, a test case's goodness measure should also consider those conditions that are on the path. 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 true 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.

We will call the magnitude of modification that is required to drive a different branch at a condition the <u>freedom space</u> 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, neither is it close to any other boundaries. 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:

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

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:

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

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.

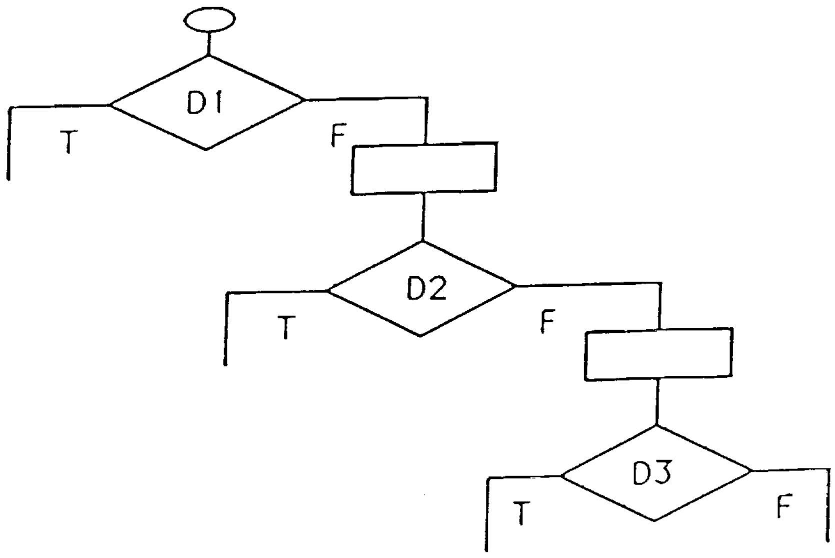


Figure 3.2.1a A sample program

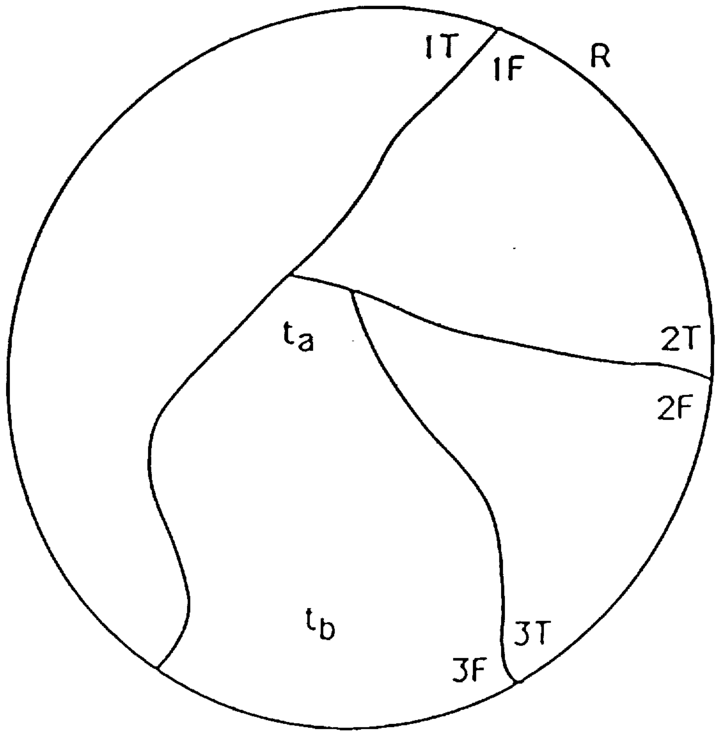


Figure 3.2.1b Input space of the program in Figure 3.2.1a

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 with the intention of covering 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 lhs(x, y, z, v_1, v_2, ...) <op> rhs(x, y, z, v_1, v_2, ...). Here, v_1,v_2,... are internal variables of the procedure. Input parameters x, y, and z may or may not be modified between the entry point and condition 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 subsections discuss some heuristics that can be used to generate new cases.

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 variables can be modified. Combinations of these modifications provide 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 showed a significant improvement over randomly generated test cases.

3.2.2.4 MODIFICATION BASED ON SYMBOLIC EVALUATION

3.2.2.4.1 BOUNDARY COMPUTATION

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

$$x_b = f1(y,z,v_1,v_2, \ldots)$$
$$y_b = f2(x,z,v_1,v_2, \ldots)$$
$$z_b = f3(x,y,v_1,v_2, \ldots)$$

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    =<  4 - 6 * z + v
```

The condition boundary will be

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

Remember that new test case generation should be based on the best case (x_1, y_1, z_1) and the modification should be as small as possible. A simple strategy would be to modify only one variable at a time. For example we can modify x and keep y and z unchanged. In this case, the condition boundary expressed for x should be used, i.e., x_b = f1(y,z,v_1,v_2, ...). In order to compute the desired value of x at D, use the actual values of y, z, v_1, v_2, ... just before D is evaluated. The computation provides the desired boundary value of x at condition D. Three new cases can be generated to cover both true and false branches: (x_b, y_1, z_1), (x_b+e, y_1, z_1), (x_b-e, y_1, z_1). Here, e is a small positive number, e.g., e = (range of x) / 100. Similarly, this case generation procedure can be applied to variables y and z.

In this procedure, an undesirable assumption is made. It is assumed that x (or y or z) would not be modified between the entry point and 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, $y_1$, $z_1$).
```

Even if the variable x has been modified, it is likely that if either y or z has not been modified, applying this test case generation procedure to one of them will be sufficient to drive the condition the other way. For this reason it is desirable to apply the procedure to all variables appearing in the conditional expression. 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 <u>inverted</u> back through the path that leads to condition D. Through this inversion, the value of x at the entry point can be found. However, this involves a complex path predicate problem which does not have a general solution [PRA87]. Heuristic approaches toward solving this problem will be presented below.

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

$$|x_b - x_c| \;/\; (2 * \mathrm{MAX}(|x_b|, |x_c|)) * 100\ \% \qquad (5)$$

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., [upper_limit_of_x - lower_limit_of_x], as its 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  $x_1 = x_c$
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.

3.2.2.4.2 FACTS USED BY THE SYSTEM

The rules accept the following types of facts:

1. (names var_name1 var_name2 ... var_namen)

   where the var_namei are the names of variables accessible to the module;

2. (types type_1 type_2 ... type_n)

   where type_i is the data type of variable i,

3. (val-at-cond test_num decision_num condition_num var_value1 var_value2 ... var_valuen)

   where var_valuei is the value of variablei at the point of the current decision, condition and test case data; and

4. (cond-expr decision_num condition_num conditional_expression)

   where the conditional_expression is in fully parenthesized infix notation.

Using these facts, the following intermediate facts are generated during execution:

1. (number-of-variables ?n)

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

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

   is used 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) in the form:

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

where var_valuei is the boundary value for variable i in the decision-condition expression for the current 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.

3.2.2.4.3 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 Test Data Generator when using Symbolic Evaluation:

500 reading-file "reads information from intermediate.results file and asserts it as val-at-cond facts"
clear-fact-list "retract invalid val-at-cond facts"
reset-read-file "reasserts read-file fact to force another read, until end of file intermediate.results
401 open-VAX-file "opens file of conditional expressions"
400 read-CE-files "reads info from desc_cond.clp and asserts it as cond-expr facts"
100 swap-right-and-left "to get current variable on left hand side of expression"

```
       do-not-swap-right-and-left "current variable is already on left hand side - this rule
          prevents the previous one from firing"
0         first-time-through "prevents Symbolic Evaluation when generating initial test case"
       open-VAC-file "to get val-at-cond information from last testing iteration's executions"
       rules to manipulate symbolic expressions, solving for the current variable
       initialize-empty-list-of-nils "to be used in creating  boundary-values facts"
       build-list-of-nils "adds one NIL for each expression"
       variable-not-in-condition "so go on to next variable"
-50       assert-boundary-expr "cond_expr symbolically solved for current variable"
       condition-not-successfully-simplified "so go on to next variable"
-100 incrementer "solve cond-expr fo next variable"
       cond-expr-done "current cond-expr processed for all variables"
-150 start-one "solve a cond-expr for its first variable"
-200 prepare-for-evaluation "get into form for variable to value substitution"
       substitute "substitute val-at-cond value for a variable name"
-250 evaluate "perform simple arithmetic to reduce right hand side"
       set-up-null-boundary-values "initialize for current val-at-cond"
-300 assert-boundary-values "replace NIL with current boundary-value for a variable"
-400 open-test-case-file "will execute every time Clips is reset"

-440 generate-midrange-initial-test-case
-450 generate-boundary-test-cases
       generate-boundary-plus-ten-percent-test-cases
       generate-boundary-minus-ten-percent-test-cases
       reassert-best-test-case "for the current decision and condition"
-460 retract-generate-initial-test-cases-fact
-470 close-test-case-file "will execute last"
```

3.2.2.4.4 CONTROL FLOW

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

3.2.2.4.5 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 "*" 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"
*       (test 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
```

3.2.2.4.6 SYMBOLIC EVALUATOR INTERFACE

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

3.2.3 TEST CASE GENERATION RULE ORGANIZATION

This research has developed many test case generation (TCG) rules. It is not desirable to use them all at once, since too many unwanted cases are generated. If one case covers a particular branch satisfying the coverage requirement, extra cases may be 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 is 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 avoids 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 = FG of BTC-i; FG = 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 additional 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 Section 3.4.

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%.

These examples demonstrate potential rule group organizations. Section 5 presents the alternatives that were 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 (i.e., driver program) that controls its execution during testing. The driver is responsible for reading the test cases from a file and passing this data to the MUT as arguments. Also, global data, out parameters, and return values are written to a file for user inspection and regression test purposes.

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. Given the instrumentation specified here, the development of an automatic instrumenter is seen to be a relatively straightforward task for those in the industry who are specializing in the design and development of Ada compilers. In fact, this could be integrated into the compiler and debugger tools in a very efficient manner. For these reasons, it was decided that prototyping of the automatic instrumentation would not be pursued immediately. However, the requirements for automatic instrumentation 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 B.

3.3.5 DIANA INTERFACE

The Parser/Scanner Module has four primary functions:

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

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

3.3.5.1 LIST OF EXECUTABLES

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

3.3.5.2 INPUT FACTS

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

Parameter Name	Type	Low Bounds	High Bounds

They are formatted as assertions to CLIPS [CLI87]:

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

3.3.5.3 DECISION/CONDITION FACTS

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

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

3.3.5.4 DIANA INSTRUMENTATION

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

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

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

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

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

would be converted to:

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

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, since they generate new test cases by randomly generating values. As implemented, these rules determine lower bounds, upper bounds, and types of the variables. A random value of the same type is generated, and the value is checked to be sure it is within the range for the variable.

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

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

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

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

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

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

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 (MUT), and the Test Coverage Analyzer (TCA). Autotest repeatedly calls the above procedures until all of the required test packets are complete. The TDG and the MUT 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 MUT. It also accumulates data for reports after the test and archives the results.

A best test case is chosen for each condition in the MUT. There can be several different methods for choosing the best test case. Currently, two methods have been implemented. The first is to calculate the distance each test case is from a border of the condition in order to select the case which is closest to the border. 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 MUT. If a condition has not been encountered before, a new entry is created in the table. If it has been encountered before, but with a different Boolean result, it is updated to indicate complete coverage. The coverage statistics are based on the number of conditions in the module under test, the number that are partially covered, and the number that are completely covered.

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

After all of the test cases for a particular packet have been viewed and used to update the coverage table, the table is searched for partially covered conditions, and the associated best test cases are returned to the test data generator. 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 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, consider a test case, (x=12, y=-8, and z=8), and a statement, IF (x > = 10) OR (y =< -10) THEN do-1 ELSE do-2. The boundary closeness measure of each individual term is calculated first. For the first term, (x > = 10), the measure is |12 - 10| / (2 * MAX(|12|, |10|) = 2/24; for the second term, (y =< -10), the measure is 2/20. The normalized measure is simply the average of these two measures. At this point earlier definitions of goodness can be applied.

3.4.2.2 TEST CASE GENERATION RULES

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

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

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

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

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

RULE-1:

IF a condition contains Boolean variables only
THEN change 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:

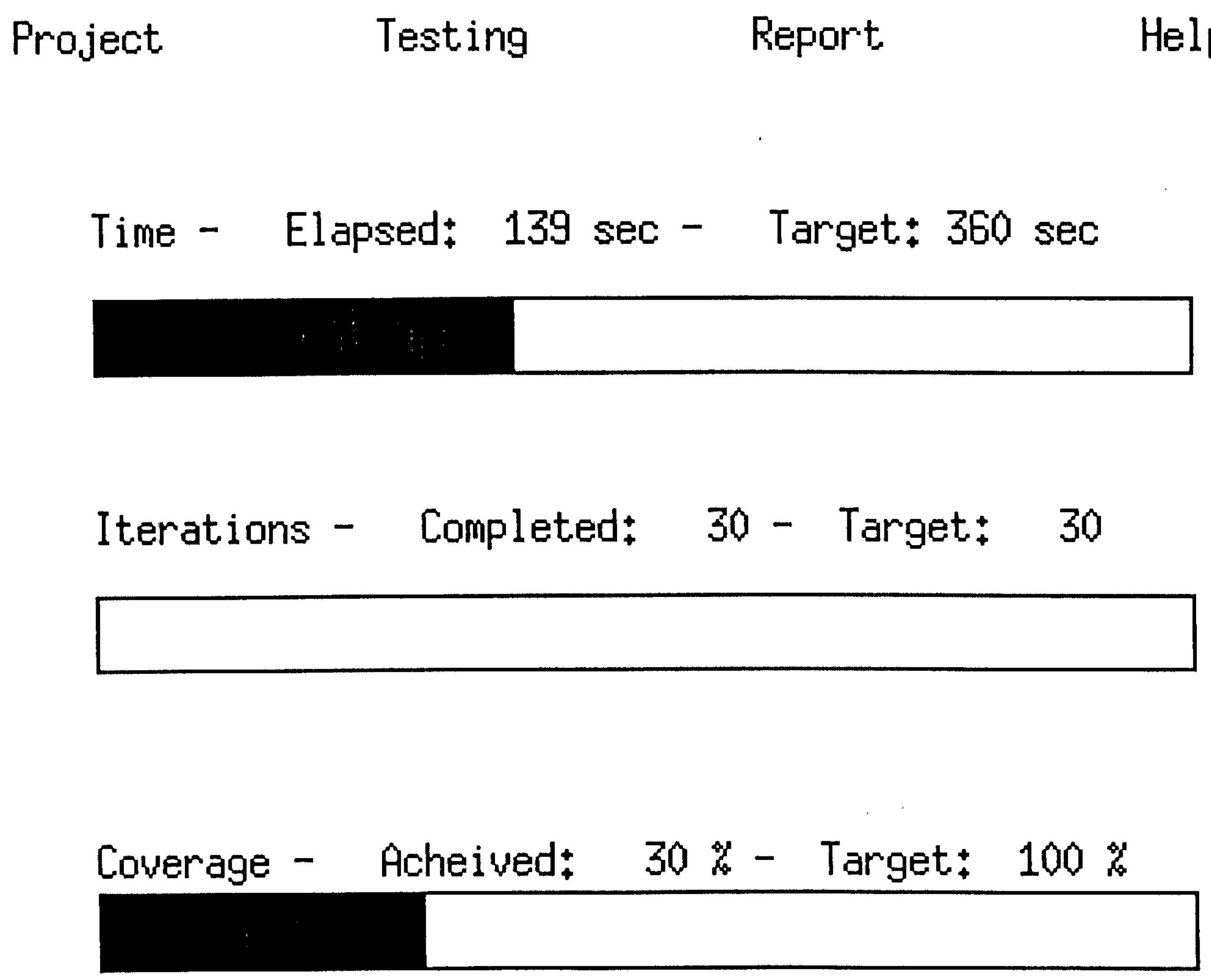
IF a condition contains both Boolean variables and non- Boolean terms
THEN
1. invert the values of the variables appearing in the input list of the best test case, one at a time, and
2. consider each Boolean condition individually and sequentially; first find the boundary, then generate cases around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

These heuristic rules will not always generate cases to cover all desired branches, but they have been shown to be an excellent starting point for multiple condition test case generation.

3.5 USER INTERFACE

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

The initial QUEST window provides the user with a number of options. As shown in Figure 3.5, the main user interface contains options for four pulldown menus: Project, Testing, Reports, and Help. The three bars on this window indicate the progress of the testing (once testing is selected). Although the bars are given initial values at the start of the application, they may be changed by selecting an option from the Testing pulldown menu.



Time - Elapsed: 139 sec - Target: 360 sec

Iterations - Completed: 30 - Target: 30

Coverage - Acheived: 30 % - Target: 100 %

Figure 3.5 User Interface Screen

3.5.1 PROJECT SUBMENU

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

When the "Project" options is selected form the User Interface Screen the pull-down menu of Figure 3.5.1 will appear. These suboptions have the following functions:

New - creates an entirely new project.
Open - allows an existing project to be opened. This will produce the window shown in Figure 3.5.1b, which gives the user the ability to select the Ada module to be tested. Entries in reverse field are subdirectories. Their selection will lead to another similar window shown in Figure 3.5.1c.
Close - closes an open project.
Delete - deletes an entire project (not enabled).
Quit - restores control to the User Interface Screen.

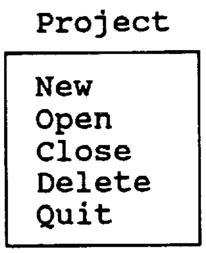


Figure 3.5.1a Project Submenu

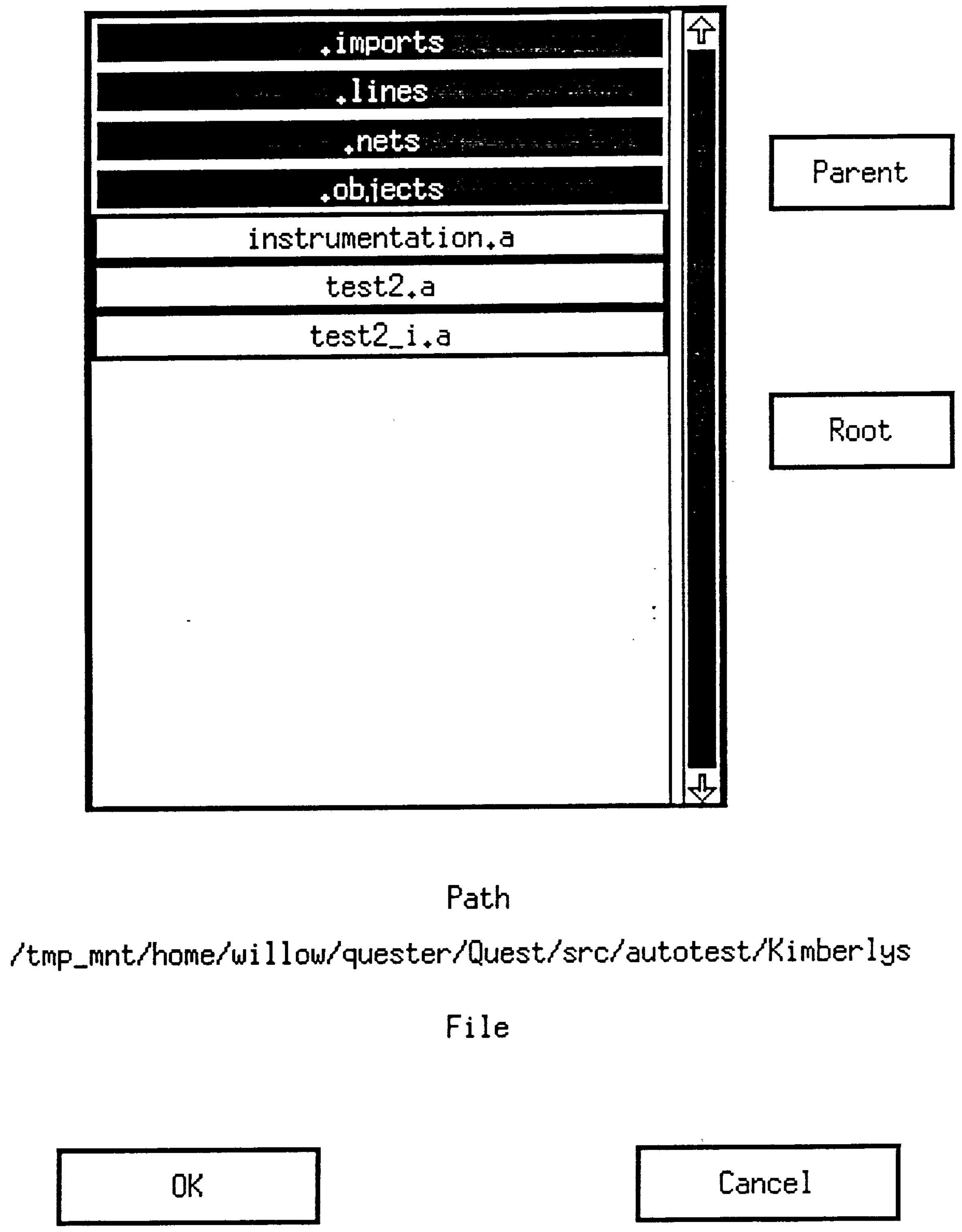
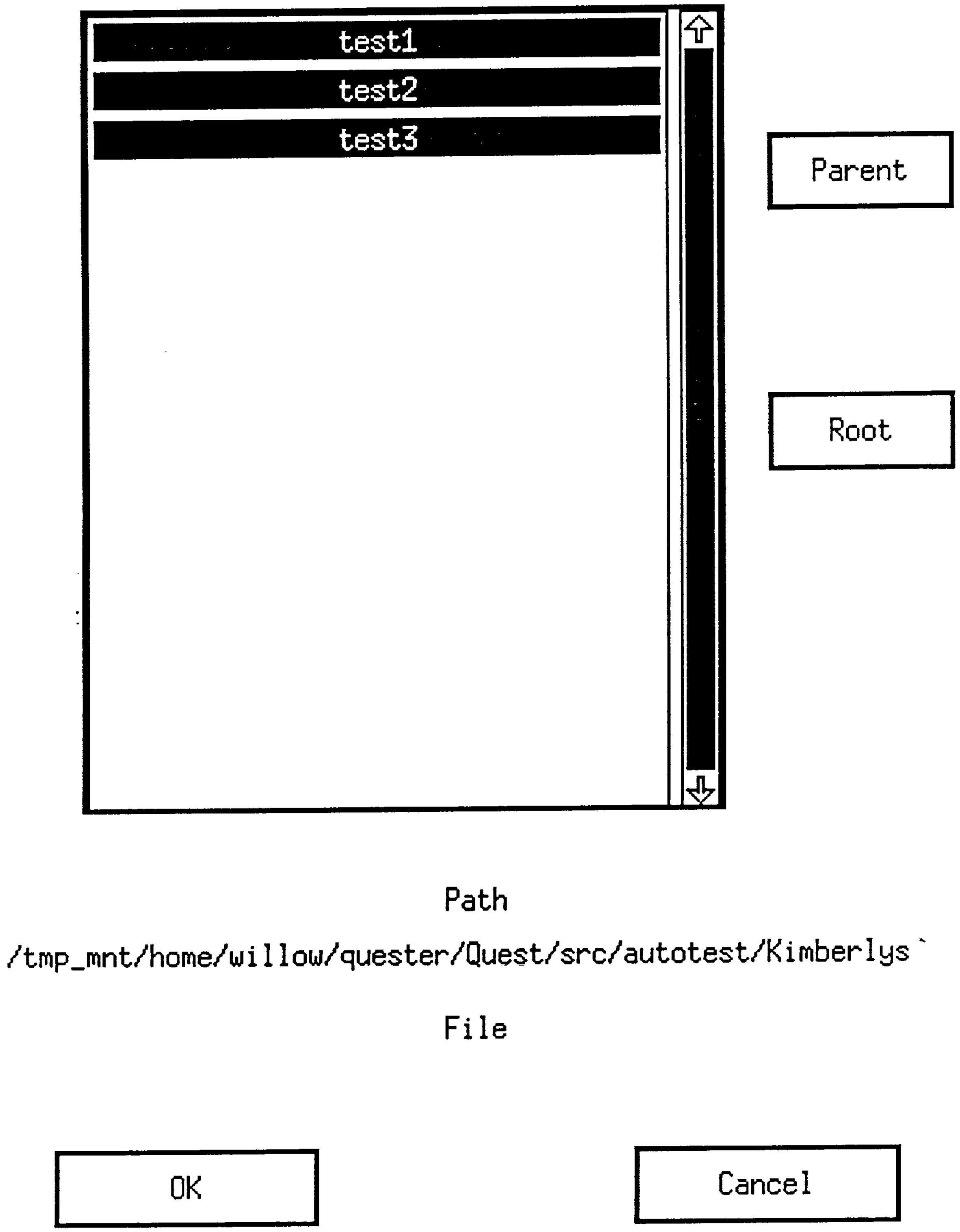


Figure 3.5.1b Directory Option Window



/tmp_mnt/home/willow/quester/Quest/src/autotest/Kimberlys

Figure 3.5.1c Subdirectory Option Window

3.5.2 TESTING SUBMENU

The Testing Section is the heart of the user interface. It allows the user to start testing, stop testing, and change the test metrics. When the "Testing"option is selected from the User Interface Screen, the testing submenu given in Figure 3.5.2a will appear.

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

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

If the values for time, iterations, or coverage given on the User Interface Screen are not desired, they may be changed through the "Set Test Metrics" option. Once selected, the window will appear which is given in Figure 3.5.2b. Any of these three values can be altered directly on the screen.

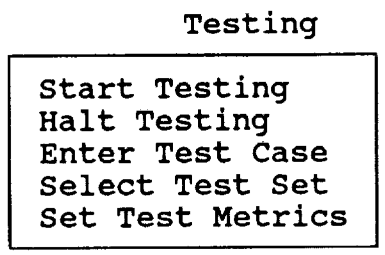
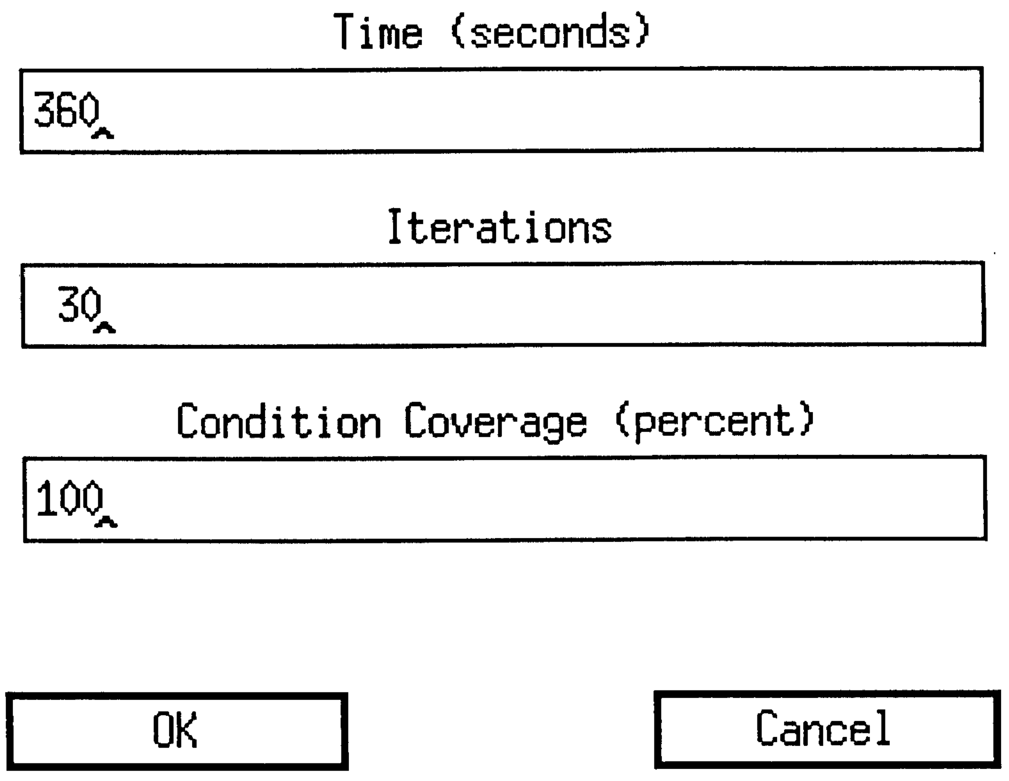


Figure 3.5.2a Testing Submenu


Time (seconds)
360

Iterations
30

Condition Coverage (percent)
100

Figure 3.5.2b Set Test Metrics Window

3.5.3 REPORTS SUBMENU

The Reports Option is used to generate reports concerning testing which has already been done. When selected, the Reports Submenu given in Figure 3.5.3 will appear. Two types of report generators are available: Coverage Reports and Best Test Case Reports. Currently, these report generators are written for the VAX VMS file management system. Conversion to the workstation environment is expected early in Phase 3.

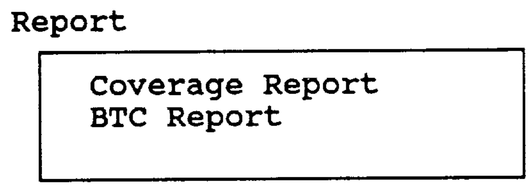


Figure 3.5.3 Reports Submenu

3.5.4 HELP SUBMENU

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

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

All of the files are collected under the same directory. For QUEST/Ada, the file names are constructed by beginning with a given system name and concatenating onto it an extension representing the data contained in the flat file. For example, if the system name was 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 onto 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 has executed without any bothersome events. If the return code is greater than zero, some event has occurred which might be important information for QUEST users (an end of file, for example). All of the return codes are 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 the file to be opened. A number of options exist to read records from the file. Before attempting any read (including the initial sequential read), call the routine lib_set_key() to tell the librarian the index file by which the data file will be indexed. Sequential reading is enabled by using two steps. First, call lib_read() with the mode LIB_FIRST_REC to rewind the offset into the index file to the first record. This will also retrieve the first record from the file, if possible. To read all records after the first, call lib_read() with the mode LIB_NEXT_REC. This can be continued until the return code from lib_read() is LIB_EOF. To read keyed files, first call lib_set_key() to set up which key and which key components are to be employed for searching. Then call lib_read() with one of two modes: LIB_FIRST_MATCH or LIB_NEXT_MATCH:

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

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

Writing records to a file is much the same. First, all of the key contents for the record must be established by calling lib_set_key() for each one. This is 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 but already exists in the index file. In this case the data record is not written to the data file.

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

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

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. The librarian routines merely take a binary collection of data and save it somewhere, leaving a method to quickly find the data again later. The librarian 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-size records. This is due to some of the information stored in the temporary files are variable length. Part of the record's information is its block number. The define LIB_BLOCK_SIZE is used to decide how much information is allocated for each block. Also included in the record is a count for how many items in the block are used. If this count equals the LIB_BLOCK_SIZE, then the next block should be checked for existence. Once the count is less than the LIB_BLOCK_SIZE define, the last block in the data is reached.

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 because the ANSI standard of memcpy() now fails when overlapping memory space is involved. The function memmove() is specifically supposed to handle copying involving overlapping memory.

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

One final portability note is that the routine vsprintf() is called 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 CONCURRENCY TESTING FOR ADA PROGRAMS

One of the main goals of concurrent program testing, debugging or analysis is to detect tasking errors -- mainly, deadlocks and concurrent access of shared variables. Like sequential programs, errors may be introduced by specifications or implementations. However, the testing of concurrent programs is more difficult to achieve. In a sequential program, a reachable error can be driven and eventually detected if the right set of input data (test cases) are provided. In a concurrent program, the testing is further complicated by the dynamic nature (and uncertainty) of the environment caused by both hardware and software factors. The hardware factors include processor locations and configuration. The software factors are attributed to the task scheduler. When a program and an environment are provided, the tester has little or no control over these factors. Because of this, these factors cannot be considered in our testing paradigm. However, if the given program is retested under the application environment, these factors should have a minimum effect.

There are two fundamental approaches to Ada concurrency testing. They are the static analysis and the task monitoring. In the static analysis, *all possible task states* of a program are explored and checked for tasking deadlocks [STR81, TAY83]. A common problem of this analysis is the unmanageable number of states. In this type of analysis, a program is not actually run; it is simply analyzed syntactically. Because a syntactically possible state may not be semantically possible, thus a large portion of the analysis effort may be wasted. In order to improve the search efficiency, a symbolic evaluation process can be integrated with the static analysis to prune semantically impossible states [TAY88]. The execution of the static analysis and symbolic evaluation can be done in serial or interleaved fashion. In the former fashion, the symbolic evaluation would not be executed until all states have been generated. On the other hand, the interleaved method allows portions of the state generation and the symbolic evaluation to be performed in turn.

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

4.1 CONCURRENCY TESTING MEASUREMENT

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

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

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

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

4.2 DATA STRUCTURES FOR CONCURRENCY TESTING

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

There are two types of coverage tables in addition to the structure described above. The first type is the summary table which lists all task units. In this table, each task unit is marked either as covered or not covered, indicating the task entry coverage. The second type of table is for each individual task unit. It contains two columns. One column indicates the task units that the titled task may call, while the other column indicates the task units that may call the titled task. Illustrations for these tables are given in Tables 4.2a and 4.2b.

Table 4.2a Task Coverage Summary Table

Task name	Covered
Task-unit-1	
Task-unit-2	X
-	
-	
-	
Task-unit-n	

Note: An 'X' indicates that a task unit has been covered. In this case, Task-unit-2 has been covered.

Table 4.2b Individual Task Coverage Table

Task-Unit-k

Call-task	Mark		Called-by	Mark
Task-unit-a_1			Task-unit-b_1	
Task-unit-a_2	X		Task-unit-b_2	X
-		-		
-		-		
-		-		
Task-unit-a_m	X		Task-unit-b_n	

Note: This example indicates that Task-unit-k has called Task-units 2 and a_m, and it has been called by Task-unit-b_2.

4.3 TOOL REQUIREMENTS

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

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

4.4 APPROACHES

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

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

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

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

4.4.1 TEST DATA GENERATION APPROACH

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

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

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

4.4.2 IRON FISTED TESTING APPROACH

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

4.4.2.1 TASK PRIORITY

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

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

2. A task which leads to a desired coverage,
3. A task which has not been executed before,
4. A task which may be called by other tasks, and the lowest priority,
5. A task which does not interact with other tasks.

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

4.4.2.2 TASK BLOCKING

A task may be blocked naturally, due to the built-in Ada scheduler, or it may be blocked artificially by the iron fisted scheduler. A naturally blocked task must be unblocked by the built-in scheduler. An artificially blocked task must be reactivated by the iron fisted scheduler. At the current stage of this project it is not clear whether it is necessary to have an iron fisted scheduler that blocks tasks. It seems that the blocking of a task will only eliminate the coverage that would have been achieved. The question of whether additional coverage can be achieved by blocking tasks is still in need of further study.

The two cases in which it is clear that artificial task blocking may be needed are: (1) there are too many tasks active, and (2) a task does not have interactions with any other tasks. The first case may help reducing the task state analysis complexity. While a task of case 2 does not have impact on the concurrency behavior of the program, task blocking may be necessary for force its instantiation.

4.4.2.3 INITIALIZATION OF TESTING

Before the iron fisted scheduler performs the tasking control, an initial execution of the program is required to provide data upon which the scheduler can function. This can be achieved by letting the program run freely for a limited time, e.g., 1 minute. During this time, the coverage information is recorded. After the time limit, the scheduler will perform the tasking control based on the achieved coverage to that point.

4.4.2.4 SCHEDULING POLICIES

The major portion of tasking control is to force a particular task selection. This will generally be required when the execution encounters a "select" statement. As mentioned earlier, task blocking may not be essential in improving coverage. If there is more than one alternative in the select statement, task priorities will be assigned to each task unit. Some scheduling examples are given in the Figures 4.4.2a-e. These will be further refined in Phase 3.

In these figures, an '*' indicates that a task has been exercised before and an 'o' indicates that a task is ready for execution (or open).

Case-1

```
Select
  |
  |---> a
  |
  |---> b              ;o
  |
  |---> c
  |
 End
```

Figure 4.4.2a If task-b is the only open task and it has not been exercised before, it will be executed.

Case-2

```
Select
  |
  |---> a              ; *, o
  |
  |---> b
  |
  |---> c
  |
 End
```

Figure 4.4.2b If task-a is the only open task, and it has been exercised before, and the same task is requesting service from task-a, then task-a will be blocked, otherwise it will be executed.

Case-3

```
Select
 |
 |---> a          ;*
 |
 |---> b          ;*, o
 |
 |---> c          ;*
 |
End
```

Figure 4.4.2c If all tasks have been exercised before, the first open task will be executed.

Case-4

```
Select
 |
 |---> a          ;o
 |
 |---> b          ;*, o
 |
 |---> c
 |
End
```

Figure 4.4.2d If multiple tasks become open at the same time, then the task that has not been exercised before will be executed.

Case-5

```
Select
 |
 |---> a          ;o
 |
 |---> b          ;o
 |
 |---> c          ;o
 |
End
```

Figure 4.4.2e If multiple tasks have become open at the same time and no task has been exercised before, then one task will be selected arbitrarily for execution.

5.0 EXPERIMENTAL EVALUATION

(Note: All figures and tables in this section appear after the narrative.)

The current prototype handles only a Pascal-like subset of Ada (i.e., no concurrency constructs). Only subprogram input parameters are considered as input to the module under test (MUT), and input is restricted to the integer and float data type. The system has been tested on various Ada programs. The results of running the prototype system on three such Ada programs are presented in this section. The Ada programs were designed to make value assignments to the variables affecting the condition branches in order to complicate complete solutions and make branch coverage difficult to determine. For example, the third Ada program was designed with function calls in the conditions, which makes symbolic determination of input values to alter the condition impossible to determine.

Figure 5.0a shows a test program graphically pretty-printed with a Control Structure Diagram (see Cross, J.H., Morrison, K.I., May, C.H., and Waddel, K.C., " A Graphically Oriented Specification Language for Automatic Code Generation (Phase 1) " Final Report, NASA-NCC8-13, Sub 88-224, September 89). Figure 5.0b shows the condition branching graph that can be abstracted from this control structure diagram. In order to experiment with rule sets reflecting various testing strategies, and to provide a basis for comparing these strategies, rules were grouped into categories consisting of the following:

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

b. Rules that take the best test cases for conditions and generate new test cases by incrementing and decrementing the input variables by a percentage, as discussed in Section 3.2.1. For results reported here, the rules increment and decrement input variables by 40% of their value rather than a percentage of their range.

c. Rules that symbolically evaluate values of input variables at conditions, finding solutions that will alter the branching, and generating new values for input variables that are clustered around these solutions. These rules implemented the ideas that were discussed in Section 3.2.2. The rule set used in these tests did not take into account conditions in the execution path leading to the condition under test and it's associated best test case. Rather, it utilized only information about the boundary value of the condition under test and generated new test cases as described next.

Given a condition, the best test case for that condition, and an input variable, the value of the variable at the boundary was symbolically determined. Then, keeping the other input variables fixed, new test cases were generated by supplying the boundary value of the input variable as input. Additionally, this value was incremented and decremented by a small amount to provide two other test cases. These rules were applied to each input variable appearing in the condition.

For each of the above rule strategies, Figure 5.0c shows the results of execution of the prototype. The table reports the test cases for which one of the three strategies found new branches in the Ada program. Because all three rule sets were initialized with the same initial test cases, the first test case covered the same path through the code. On the third test case, for example, the table shows that random test case generation rules found another path through the code adding three more branches to the number of branches covered by this rule set.

Figure 5.0d graphically presents the information from the table with the y axis showing the percentage of the 20 branches which were covered by the test cases. Note that the x axis is not a linear scale, and as such, the graph does not accurately reflect the relative speed with which maximum coverage is achieved.

Figure 5.0e and f show the code and flow graph for the second Ada test program, and Figure 5.0g presents the results of the three testing methods. In this test, complete branch coverage was achieved by the symbolic evaluation rule set in 21 test cases. The increment/-decrement rule set found the final branch on the 87th test case, and random test case generation was considerably worse. In the flow graph, the conditions have been numbered as they were during testing and combined with Figure 5.0h, which shows the actual conditions covered following the conclusion of the testing. We can see, for example, that random testing failed to find the true branch of condition three, and consequently could not cover conditions four and five. Random testing also failed to uncover the true branch of condition six before each of the other rule sets had found complete branch coverage.

Figure 5.0i shows the percentage coverage graph for the second Ada program. For the third Ada program, the code and flow graph are given in Figure 5.0j. As in previous examples, Figures 5.0k, l and m show the results of testing this code with the three rule sets. In all three test programs, the rule based testing using heuristic rules reflecting increment/decrement or symbolic evaluation strategies outperformed random test case generation. The symbolic evaluation strategy did not always outperform the increment decrement strategy, but when it failed (in test three) to obtain coverage for the number of tests run, it appeared to be only slightly behind the increment decrement strategy. The reason for failure in test three using symbolic evaluation was the call to the user function as part of a condition. Since the symbolic evaluator has no idea what the function does, it is unable to successfully determine a branching condition.

```
        *** GRASP/ADA V1.0         File: test1.a.csd   Page:      1

with text_io, instrumentation;
use text_io;

procedure test1( w:  in out integer; x:  in out integer; y:  in out integer;
   z:  in out integer) is

   function user_f( a : integer) return integer is

      temp: integer;
   begin
      temp := a / 100;
      return (temp);
   end user_f;
begin
   if x > 5 then
      while x > 5 loop
         x := x - 256;
         z := z + 10;
      end loop;

   elsif x = 1 then
      if z < x then
         z := z + 20;

      else
         z := x;
         x := x - 1000;

      end if;
      if x <  -100 then
         x := x + user_f(z) + y;

      else
         z := z + 100;
         x := user_f(z) * y;

      end if;

   else
      x := user_f(z) * y;

   end if;
   if y < 300 then
      z := z * 2;
      x := x * 9;

   else
      z := z * 3;
      x := x * 12;

   end if;
   if 2 * y + z + 6 then
      if w > 100 then
```

Figure 5.0a   Ada Source Code for Test Program 1

```
      *** GRASP/ADA V1.0        File: test1.a.csd   Page:      2
│ ││  ││├── x := x + 10;
│ ││  ││├─◇┐if w > 90000 then
│ ││  │││ │├── y := y + 2 - 100;
│ ││  │││ │└
│ ││  │││ └┐else
│ ││  │││  ├── y := y + 2 - 100;
│ ││  │││  └
│ ││  │││   end if;
│ ││  ││└
│ ││  │└┐else
│ ││  │ ├── x := user_f(y) + z;
│ ││  │ └
│ ││  │  end if;
│ ││  └
│ ◇┐elsif y > 10000 then
│ │├── y := y - 10000;
│ │└
│   end if;
└end test1;
```

Figure 5.0a   Ada Source Code for Test Program 1

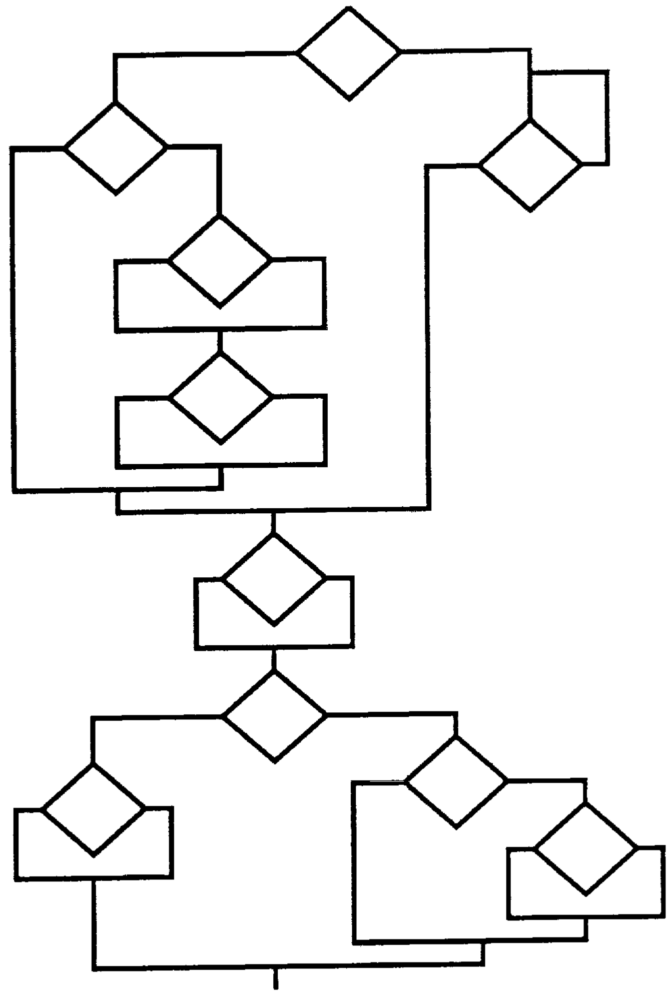


Figure 5.0b Flow Graph of the First Test Program

Test Case	Inc/Dec by 40%	Random	Symbolic
1,	5,	5,	5
3,	5,	8,	5
7,	5,	8,	8
8,	5,	9,	8
15,	8,	9,	8
21,	8,	10,	8
27,	8,	10,	11
52,	8,	10,	12
56,	8,	10,	13
76,	8,	10,	15
135,	8,	10,	17
232,	8,	10,	19
870,	11,	10,	19
1105,	13,	10,	19

Figure 5.0c Results for the First Test Program in Table Form

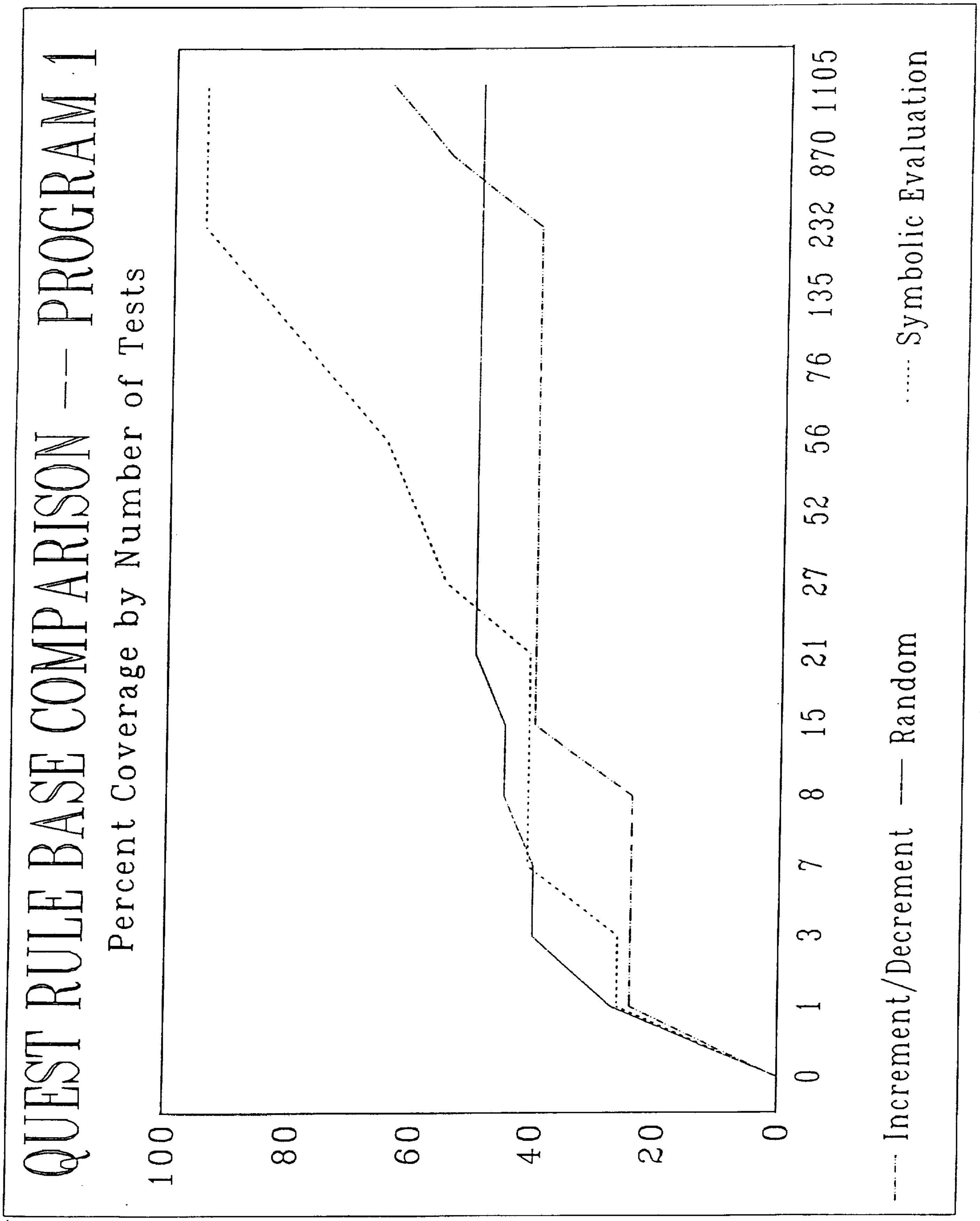


Figure 5.0d Results for the First Test Program in Graphical Form

```
      *** GRASP/ADA V1.0         File: test2.a.csd   Page:     1

with text_io, instrumentation;
use text_io;

procedure test2( x:  in out integer; y:  in out integer; z:  in out integer)
   is

   t1, t2 : integer;

   function user_f( a : integer) return integer is

      temp: integer;
   begin
      temp := a / 100;
      return (temp);
   end user_f;
begin
   x := x - 100;
   t1 := y - 100;
   t2 := z - 100;
   while x > 1 loop
      x := z - 2000;
      if y < 10 then
         if y >  -5 then
            y := y + t2;
            loop
               z := z +  abs (y);
               t1 := 2 * y;
               if z > 19950 then
                  y := y + 10;

               else
                  y := y - 10;

               end if;
               z := z - 1000;
               exit when z > 20000;
            end loop;

         else
            if y >  -20 then
               z := z - 10000;

            else
               z := z - 20000;

            end if;

         end if;

      end if;
      y := x + z + user_f(t2);
      z := user_f(y) + user_f(t1);
   end loop;
   if y < 100 then
```

Figure 5.0e Ada Source Code for Test Program 2

```
│ ││├── y := y + 100;
│ ││└
│   end if;
└end test2;
```

Figure 5.0e Ada Source Code for Test Program 2

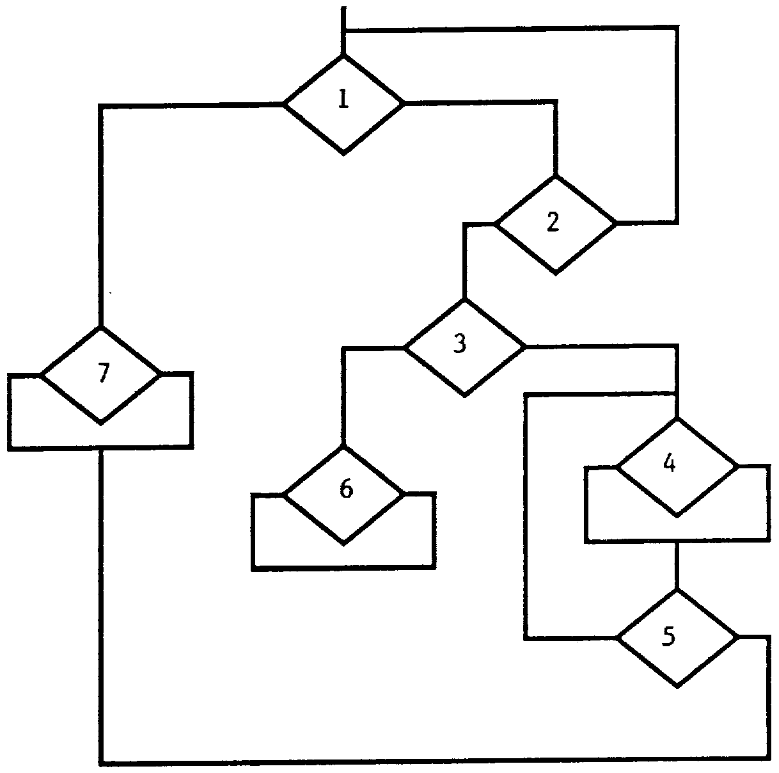


Figure 5.0f Flow Graph of the Second Test Program

Test Case	Inc/Dec by 40%	Random	Symbolic
1,	9,	6,	9
3,	9,	7,	11
4,	9,	8,	11
9,	12,	8,	13
12,	13,	8,	13
21,	13,	8,	14
87,	14,	8,	14

Figure 5.0g Results for the Second Test Program in Table Form

Test Two (7 conditions, 14 branches)

	1t	1f	2t	2f	3t	3f	4t	4f	5t	5f	6t	6f	7t	7f
Random	*	*	*	*		*						*	*	*
Inc Dec	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Sym Eval	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Figure 5.0h Condition Coverage at End of Testing for Second Test Program

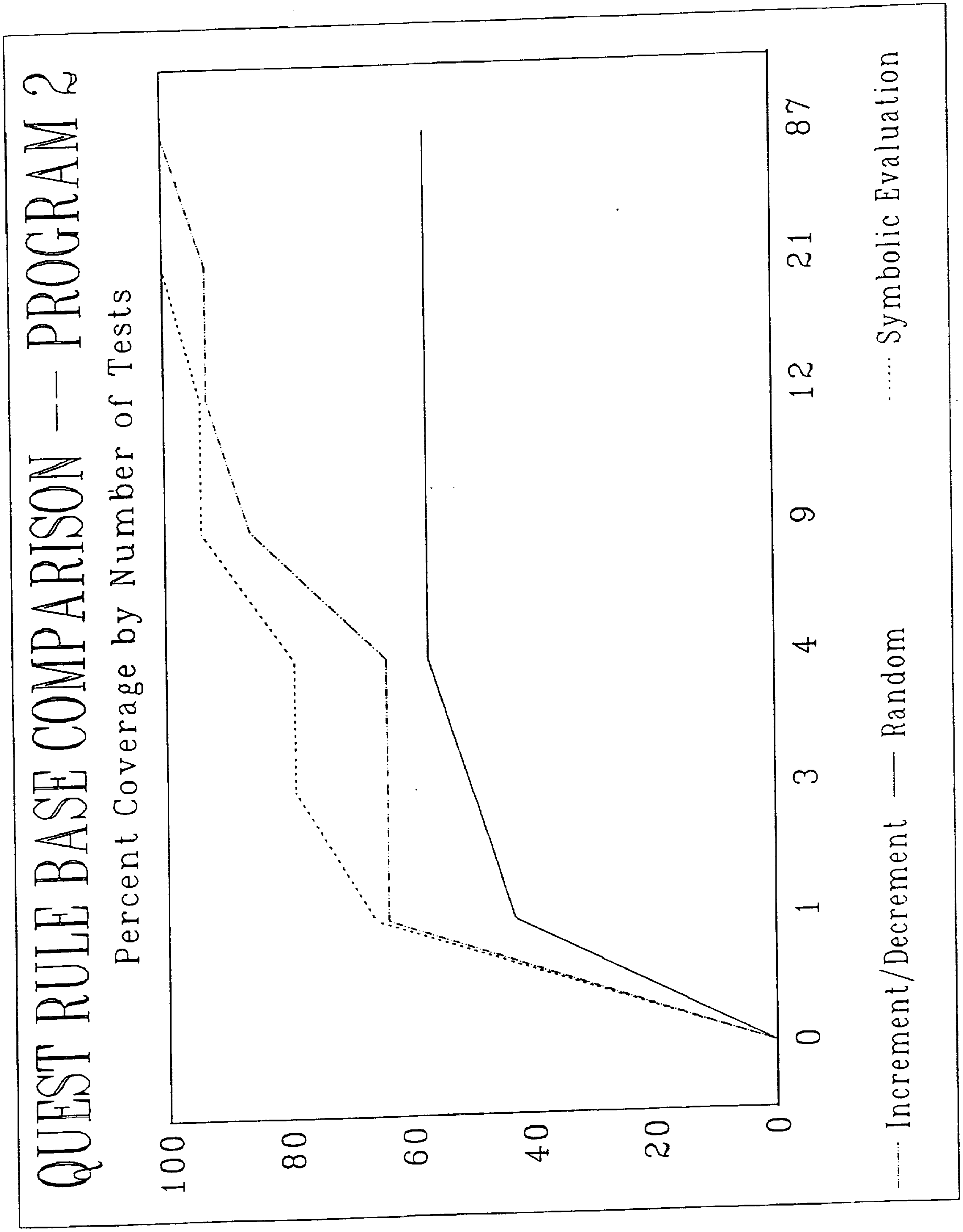


Figure 5.0i Results for the Second Test Program in Graphical Form

```
     *** GRASP/ADA V1.0          File: test3.a.csd    Page:      1

procedure test3( a:  in out integer; b:  in out integer; c:  in out integer;
   d:  in out integer) is

   function user_f( a : integer) return integer is

      temp: integer;
   begin
      temp := a / 100;
      return (temp);
   end user_f;
begin
   if c = d then
      d := a * b + c;
      if b < d then
         b := b - c;
         d := b - d;
         if c > 9950 then
            c := c * 100;
            b := b - d;
            d := 2 * b * 3 * c;

         else
            c := c * 400;
            d := 2 * b * 2 * c;

         end if;

      else
         b :=  -1 * c;
         d := d + b;

      end if;
      b := 0;
      if a > 5 then
         if a < user_f(1300) then
            if a > user_f(700) then
               a := a - d - user_f(c - 50);

            else
               a := a + d + user_f(c + 50);

            end if;

         else
            a := a - d - user_f(c - 100);

         end if;

      else
         a := a + d - user_f(c);

      end if;
```

Figure 5.0j Ada Source Code for Test Program 3

```
      *** GRASP/ADA V1.0        File: test3.a.csd   Page:      2

    else
       d := a + b;
       if b > 100 then
          b := 2 * c;
          d := d + b;

       else
          b := 1000;
          d := d + b;
          if c > user_f(9950) then
             c := c + 105;
             d := 2 * b * 2 * c + 3;

          else
             c := c - 105;
             d := 2 * b * 2 * c - 3;

          end if;

       end if;
       if a > 0 then
          a := a - b + user_f(d);

       else
          a := c + d;
          if a =  -100 then
             a := a - b - user_f(d - 50);

          end if;

       end if;

    end if;
end test3;
```

Figure 5.0j Ada Source Code for Test Program 3

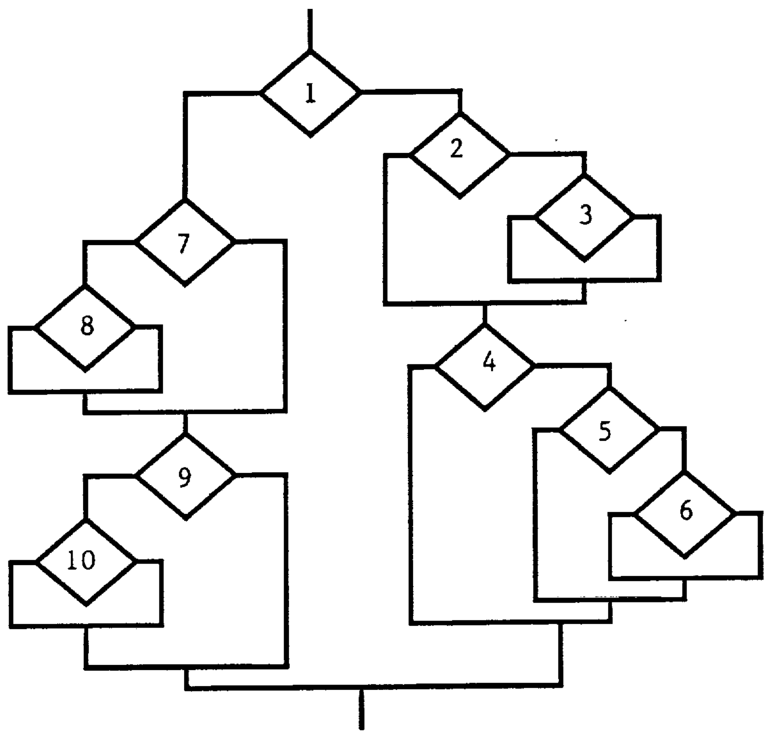


Figure 5.0j Flow Graph of the Third Test Program

Test Case	Inc/Dec by 40%	Random	Symbolic
1,	3,	3,	3
2,	3,	5,	3
3,	3,	6,	3
6,	3,	8,	5
9,	3,	8,	10
11,	6,	8,	10
12,	11,	8,	10
14,	14,	8,	10
35,	14,	8,	12
85,	14,	8,	13
108,	14,	8,	14
116,	15,	8,	14
240,	16,	8,	14

Figure 5.0k Results for the Third Test Program in Table Form

Test Three (10 condition, 20 branches)

	1t	1f	2t	2f	3t	3f	4t	4f	5t	5f	6t	6f	7t	7f	8t	8f	9t	9f	10t	10f
Random		*											*	*	*	*	*	*		*
Inc Dec	*	*	*	*		*	*	*	*	*	*	*		*		*	*	*		*
Sym Eval	*	*	*	*		*	*	*		*			*	*		*	*	*		*

Figure 5.01 Condition Coverage at End of Testing for Third Test Program

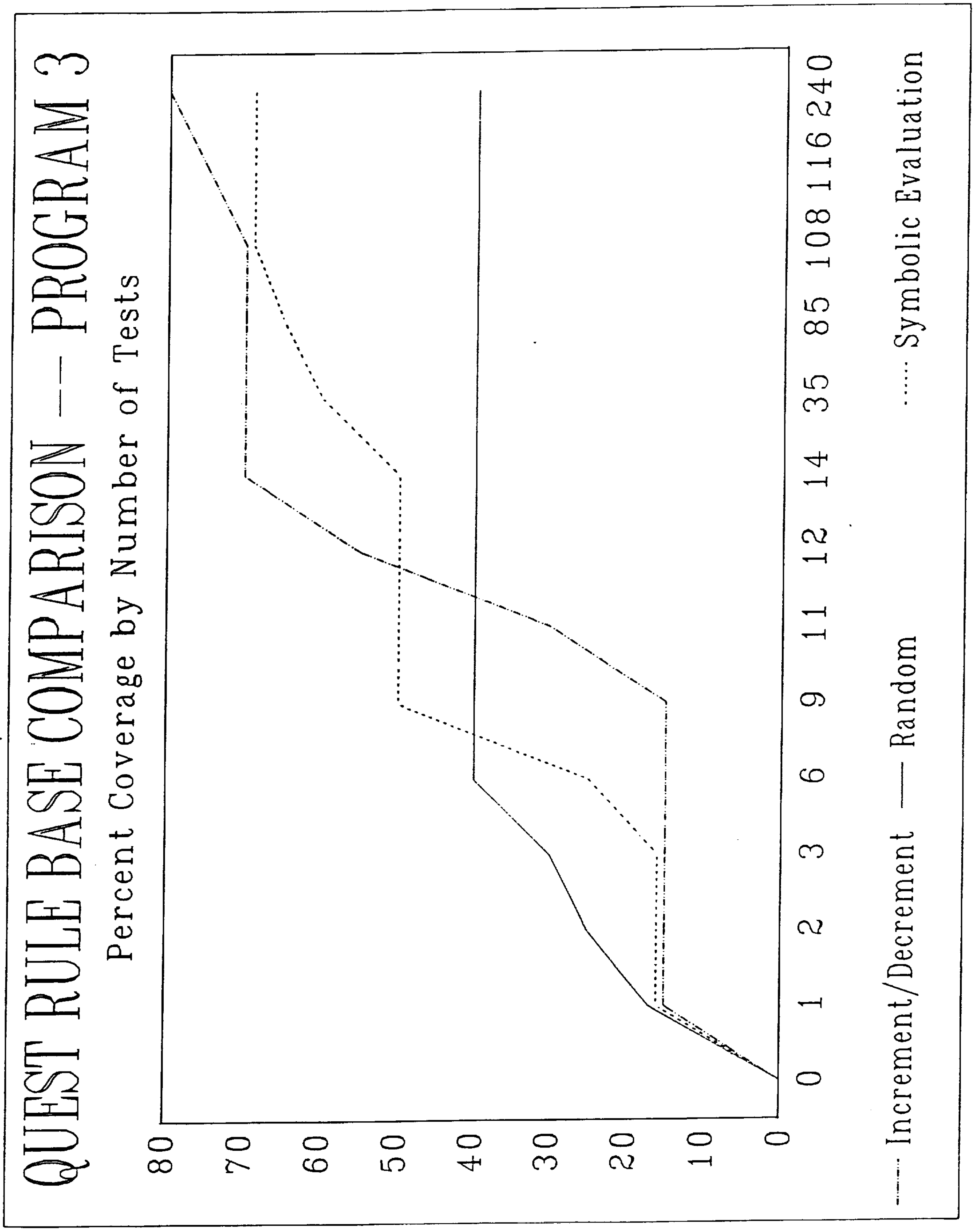


Figure 5.0m Results for the Third Test Program in Graphical Form

6.0 PROJECT SCHEDULE

6.1 REVIEW OF PROJECT GOALS

The primary goals of software support tools for Ada are to improve software quality and reliability as well as increasing development efficiency. Phase 1 of the current project designed and prototyped an environment to facilitate expert system assisted testing of Ada code. A formal grammar specification of Ada and a parser generator were used to build a preliminary Ada source code instrumenter. A prototype rule base was developed using the CLIPS [CLI87] expert system tool, and the prototype performed test data generation on instrumented Ada programs using a feedback loop between a test coverage analysis module and an expert system module. The expert system module generated new test cases based on the information provided by the analysis module. The result of Phase 1 was the demonstration of the feasibility of the rule-based unit level testing paradigm through a complete system prototype.

Phase 2 of the project designed and evaluated the use of more sophisticated rule bases, formed a preliminary design for a concurrency testing approach, and established contacts with NASA subcontractors currently developing Ada software. A more sophisticated rule base developed to model Ada programs as abstract functions was developed using symbolic evaluation of the source code conditions. This rule base has the ability to handle several mathematical programming constructs more efficiently than the original prototype rule base, which was itself refined. The interaction of the two rule bases has been studied to develop rules to further optimize the expert system module. A preliminary design for concurrency testing has been developed, analyzed, and refined. On December 5, 1989 a meeting was held with Mr. Ken R. Dunne, Principal Engineer of Boeing Aerospace to evaluate the needs of the NASA subcontractor's programming environment. The need to develop a workstation prototype was discussed, as well as the applicability of various automated testing approaches to the Boeing application environment.

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

6.2 RESEARCH APPROACH

This phase of the research includes the following subtasks.

1. **Refinement of the rule base.**

The symbolic evaluation rule base developed in Phase 2 is capable of generating test data for unit-level testing which obtains coverage that is very difficult to achieve with standard test case generation. However, it is clear that modifications of the rule base could lead to even better sets of test cases (i.e., those which lead to greater coverage, or the same coverage with fewer test cases). Refinements to the symbolic evaluation rule base will be developed and implemented using the CLIPS expert system tool in an attempt to establish rule bases which improve upon the current system. Additionally, new rule-based testing methods will be evaluated for effectiveness in generating additional unit-level coverage.

2. Completion of the rule base evaluation.

The comparative evaluation of the various rule-based testing strategies will be continued and completed. The completed evaluation will qualitatively discuss the strengths and weaknesses of each rule and rule type, and will quantitatively present the effects of interacting rule bases on the unit-level coverage of a variety of Ada programs. The capstone of the quantitative evaluation will be an attempt to identify an optimal general rule base across a cross-section of testing paradigms and programs.

3. Implementation and evaluation of a concurrency testing prototype.

A preliminary design for a concurrency testing prototype has been developed in Phase 2 of the project, and this design will be refined and implemented using concurrency tasking information provided by the Verdix DIANA Ada interface package. The concurrency testing approach will provide current history coverage information through the use of the "iron-fist" task scheduling monitor which will force determinism into the Ada rendezvous by locking all but one possible rendezvous in the case of multiple rendezvous selects. This approach, combined with the unit-level testing approach already developed in Phases 1 and 2, will initially cover all rendezvous. The rendezvous coverage prototype will be evaluated to determine the possibilities of extending the coverage metric to a more general case, such as Taylor task histories.

4. Development of a workstation-environment prototype.

The meeting conducted with Boeing during Phase 2 of the project indicated the need for a workstation environment for the testing prototype. This workstation environment will be developed in Phase 3 of the project. In addition to providing a new user interface which reflects current user interface design techniques, the development of the workstation prototype will afford the opportunity to expand the features of the prototype. One important expanded feature will be the use of the Verdix DIANA Ada interface package in the place of the current attributed grammar in the parser/scanner module. It is projected that the use of the DIANA interface will provide advantages in the development of the concurrency prototype and the transition from prototype to working package, as well as making QUEST compatible with the APSE standard.

5. **Development of a technology-transfer document.**

In order to speed the transfer of technology from the research environment to a working environment, a prototype development document will be created. The prototype development document will be a concise overview and a detailed explanation of each module of the prototype system. It will also contain descriptions of the directions which might be taken to expand the prototype modules into a more robust system. The purpose of this document will be to allow any interested NASA subcontractor to quickly develop a robust working automated testing environment from the prototype developed during this research.

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

The contacts established with Boeing in Phase 2 of this project provided useful insight into the requirements NASA subcontractors have for an automated program testing tool. Continued interaction with these contacts will aid in the development of a concurrency testing prototype appropriate to existing concurrent Ada software. We also look to them for direction in the development of the technology-transfer document.

6.3 PROPOSED RESEARCH SCHEDULE

The Gantt chart in Figure 6.3 provides the sequence of Task I activities to be accomplished during Phase 3 of this project. Details for Phase 3 activities are presented above.

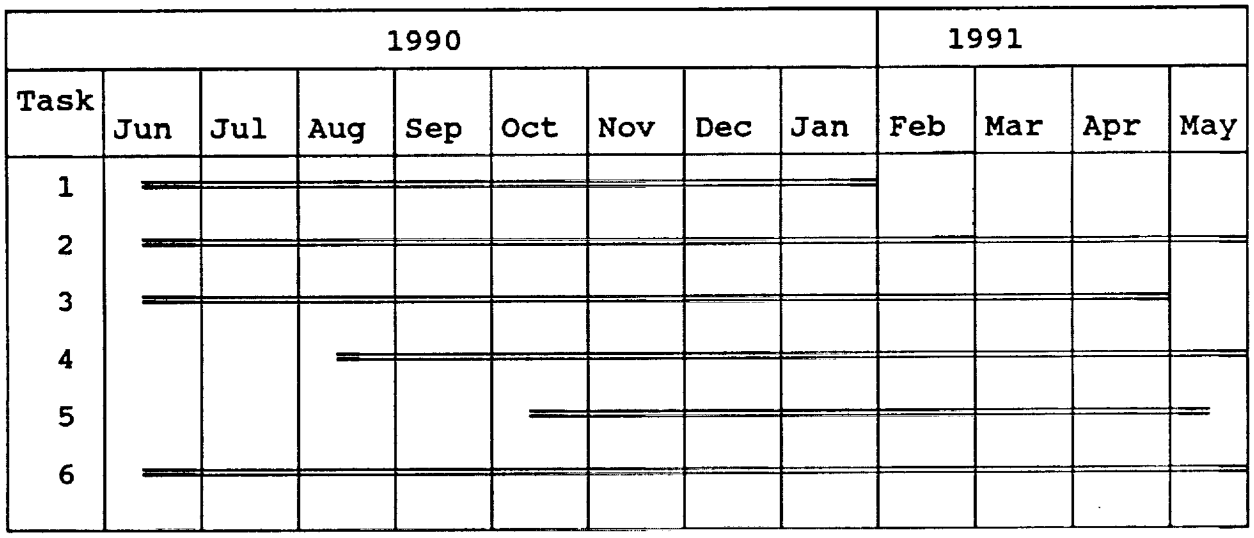


Figure 6.3. Phase 3, Task I Gantt Chart

7.0 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 PROCEEDINGS OF 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/Ada[1] 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/Ada[1] 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]

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

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 "imput 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. (upper-limit - lower-limit)/2, lower and upper values.

Best test case selection

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

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

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 c1,c2 precede condition c3 in the execution path, and t1,t2,t3 represent the "closeness" values associated with a test case t, then for weights w1,w2,w3 a value determined by

$$w3*t3 + w2*(1/t2) + w1*(1/t1)$$

represents a better measure of the test case than does the value t3. Note that the values of t1,t2,t3 are in [0,1].

In general, if c_1, c_2, ... 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, ... w_n, the best test case for condition n is chosen by a minimum value of

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

For testing in QUEST, weights of 1 for w_n and 1/(n-1) for $w_1...w_{n-1}$ 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)))
    (fprintout test-case-file crlf)
;next test case
    (bind ?outer_pointer (+ ?outer_pointer 1)))
)
```

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

Listing 2.

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

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

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

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

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

Listing 3.

```
(defrule generate_symbolic_approximation_plus_increment_test_cases ""

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

CONCLUSION

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

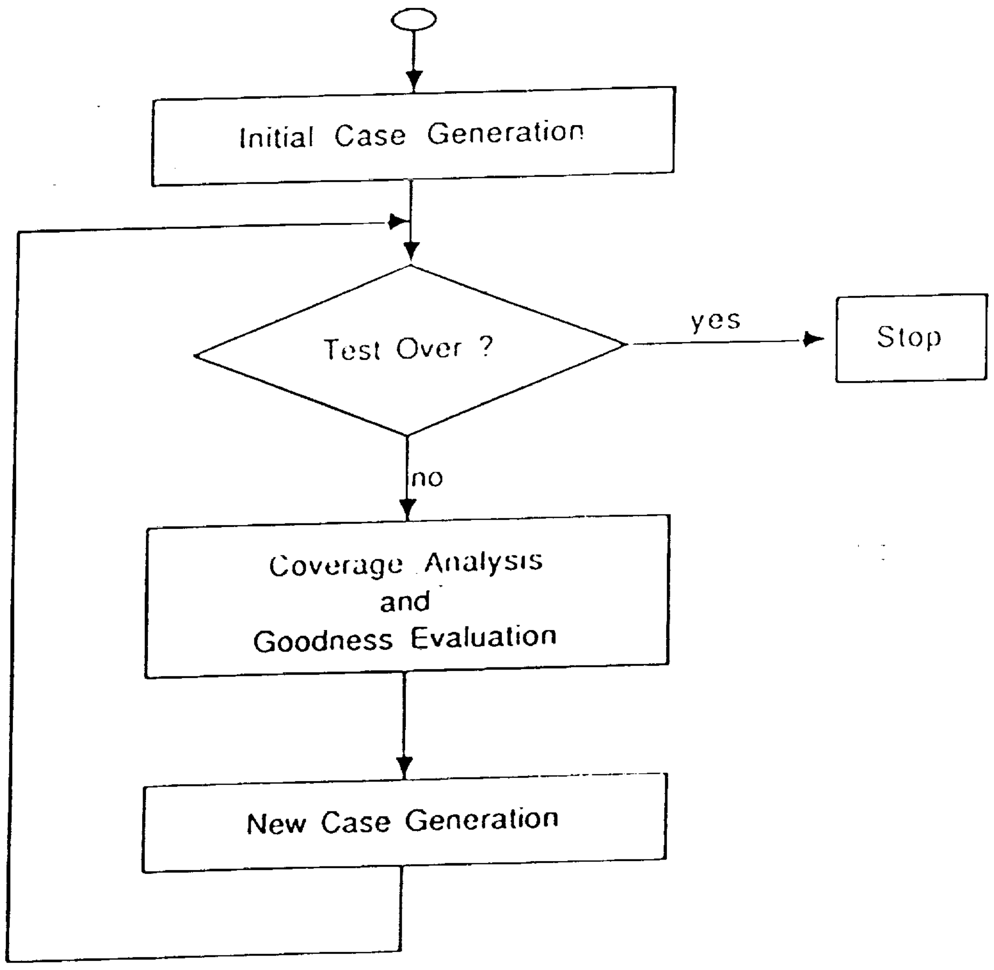


Figure 1 System Concept of the Intelligent Test Data Generator

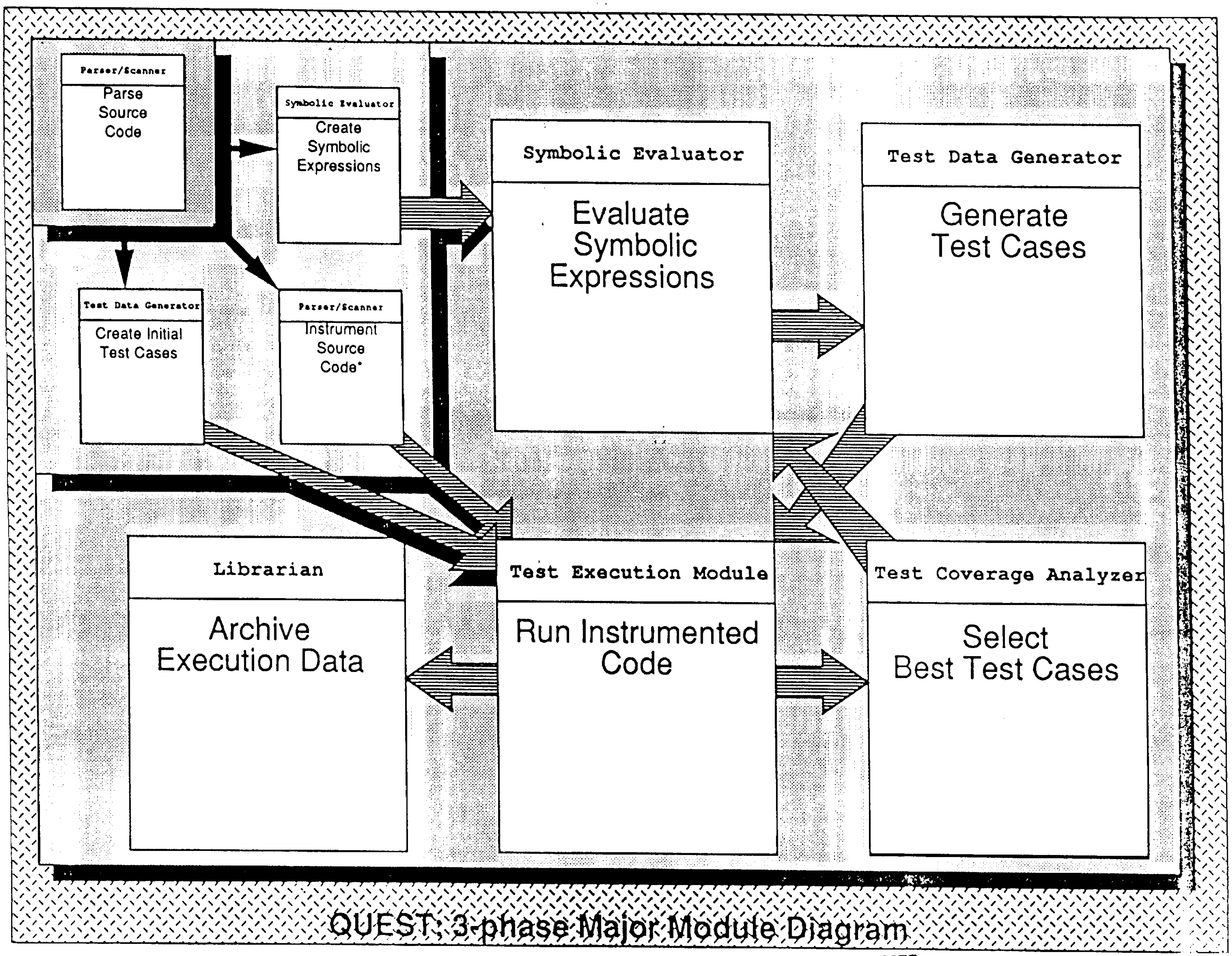


Figure 2

PRECEDING PAGE BLANK NOT FILMED

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;
  side1,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, side1);
    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) or 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,parm1,parm2,...
      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,relop(TestNum,3,1,side1+side2,LT,side3)) then
          return_val := 0;
        elsif decision(TestNum,4,relop(TestNum,4,1,side1,EQ,side2))
                                                            then
          return_val := 5;
        else
          return_val := 4;
            end if;
        elsif decision(TestNum,5,relop(TestNum,5,1,side1,GT,side3)
                            or relop(TestNum,5,2,side2,GT,side3)) then
          return_val := 0;
        elsif decision(TestNum,6,relop(TestNum,6,1,side1,EQ,side2))              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)
                            and relop(TestNum,8,2,side2,GT,10)) then
            return_val := return_val + 10;
        elsif decision(TestNum,9,relop(TestNum,9,1,side1,GT,1) and
                            relop(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,parm1,parm2,...
        get(indata,side1);
        get(indata,side2);
        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,
   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,relop(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,relop(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,parm1,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,relop(TestNum,1,1,a,LT,1)) loop
      if decision(TestNum,2,relop(TestNum,2,1,a,GT,b)) then
        c := 713 mod a;
        while decision(TestNum,3,
                        relop(TestNum,3,1,c,GT,a)) loop
          c := c - 2;
          d := d - 1;
          if decision(TestNum,4,
                        relop(TestNum,4,1,c,GT,d)) then
            d := d-2;
          else
            null;
          end if;
          if decision(TestNum,5,
                        relop(TestNum,5,1,c,LT,b)) then
            if decision(TestNum,6,
                          relop(TestNum,6,1,c,LT,213 mod b)) then
              if decision(TestNum,7,
                            relop(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,relop(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 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,relop(TestNum,1,1,j,LT,50)) loop
         if decision(TestNum,2,relop(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,relop(TestNum,3,1,i,LE,k-3)) loop
         i := i+3;
      end loop;

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

```
              return(3);
            else
              return(4);
            end if;
          end if;
        elsif decision(TestNum,7,relop(TestNum,7,1,X,LT,10.0) and
relop(TestNum,7,2,Y,GT,10*Z)) then          if
decision(TestNum,8,relop(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,parm1,parm2,...
      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.

APPENDIX D. PAPER: A HEURISTIC APPROACH FOR TEST CASE GENERATION BY CHANG, CARLISLE, CROSS AND BROWN SUBMITTED TO THE 1991 ACM COMPUTER SCIENCE CONFERENCE

A HEURISTIC APPROACH FOR TEST CASE GENERATION[*]

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

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

ABSTRACT

Test case generation using traditional software testing methods generally requires considerable manual effort and generates only a limited number of test cases before the amount of time expended becomes unacceptably large. A rule-based framework that will automatically generate test data to achieve maximal branch coverage is presented. The rationale of the heuristic rules and the strategy for the test case generation are also described. The result of this approach shows its potential for improving software testing. The rule-based approach allows this framework to be extended to include additional testing requirements and test case generation knowledge.

Keywords: Software Testing, Rule-based Systems, Software Engineering, Artificial Intelligence.

[*] This work is supported, in part, by George C. Marshall Space Flight Center, NASA/MSFC, AL 35812 (NASA-NCC8-14).

1. INTRODUCTION

The main thrust of software testing research has focused on the development of formal methods of software and system testing [BEI83]. By definition, "testing .. is the process of executing a program (or a part of a program) with the intention or goal of finding errors" [SHO83]. A test case is a formally produced collection of prepared inputs, predicted outputs, and observed results of one execution of a program [BEI83]. Program testing methods can be classified as either dynamic analysis, static analysis, or a combination of these [RAM75]. Dynamic analysis of a program involves executing the program with test cases and analyzing the output for correctness, while static analysis includes such techniques as program graph analysis and symbolic evaluation. A dynamic test strategy is a method of choosing test case from the functional domain of a program. It is based on criteria that may reflect the functional description of a program, the program's internal structure, or a combination of both [ADR82]. These criteria specify the method of test case generation to be used for a dynamic test strategy.

Generating test inputs to a program may not appear to be a difficult problem since it may be done by a random number generator [DUR81]. However, random testing alone has been shown to be an inadequate method for exposing errors. When combined with extremal and special value (ESV) testing, it can be an effective method and can provide a direction for the generation of future test cases [VOU86]. On the other hand, algorithms for generating test case to satisfy particular adequacy criteria have generally poor time and space complexities and produce small amount of test cases.

The objective of this research is to design and develop a framework that will automatically generate test cases to achieve maximal branch coverage of an arbitrary program. A rule-based approach allows this framework to be extended to include more testing requirements and test case generation knowledge. Detailed report on this research can be found in [BRO90].

2. FRAMEWORK

The outline of the framework is shown in Figure 1. It is divided into four major components: parser/scanner, test case generator, test case analyzer, and report generator.

2.1 PARSER/SCANNER

The purpose of the parser/scanner is to instrument the source code and create a file containing information about the source code's structure. The instrumentation of the code is done by inserting a function call into the condition part of an IF-THEN statement. For example, statement IF (8*Y-4*X+5) >= (5-Z) THEN do-1 ELSE do-2 will be instrumented as IF analyze((8*Y-4*X+5) >= (5-Z)) THEN do-1 ELSE do-2. "Analyze" is a function defined in the test case analyzer which performs coverage analysis and test case evaluation (see the section of test case analyzer). The structure file provides the test case generator information about the code so that test cases can be generated accordingly. The main concern of the code structure is the IF-THEN statement. An IF-THEN statement is recognized as IF LHS <op> RHS THEN do-1 ELSE do-2, where RHS and LHS stand for right-hand-side and left-hand-side of the condition expression, respectively. "<op>" denotes a logical operator such as <, >, =, =<, >=, or <>.

2.2 TEST CASE GENERATOR

The test case generator produces new test cases that would drive (or cover) target branches/conditions in the code. It takes coverage results from the analyzer and code information from the structure file and determines what conditions/branches should be targeted for new case generation. It then uses heuristic rules to generate more test cases. When a set of new cases is generated, it is stored in a test case file.

2.3 TEST CASE ANALYZER

The test case analyzer runs the new test cases in the instrumented code, records the cumulative branch and condition coverage, and evaluates the "goodness" of each test case. The analysis result is then fed back to the test case generator for further case generation.

2.4 REPORT GENERATOR

When the test case generation and analysis cycle is completed, the report generator will print the results for the user. The information to be printed in the report can be specified by the user. These include test cases, condition and branch coverage, and statistics of the cases and coverage.

3. TEST CASE GENERATION STRATEGY

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

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

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 2b, will not be targeted for new case generation. This is because no test case model yet exists that can be used for modification. A test case model will eventually surface later in the process, and in this example, after condition 5 is fully covered, a model for condition 6 will appear.

3.1 BEST TEST CASE

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

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

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

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

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

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

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

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

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

where:

G(t,D) : Goodness of test case t at condition D.

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

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

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

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

3.2 TEST CASE GENERATION PROCEDURE

The basic idea of new case generation is to modify the best test case of a target condition slightly with the intent to drive the uncovered branch of the condition. In Figure 4, input to the procedure contains three parameters x, y, and z. Assume condition D's truth branch is covered, and its best test case is (x_1, y_1, z_1). More cases must be generated to cover D's false branch. Condition D can be expressed as LHS(x, y, z, v_1, v_2, ...) <op> RHS(x, y, z, v_1, v_2, ...). Here, v_1,v_2,... are internal variables of the procedure. Input parameters x, y, and z may or may not be modified between the entry point and condition D. The following sections discuss some approaches that have been used to generate new cases.

3.2.1 INCREMENT AND DECREMENT MODIFICATION

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

3.2.2 BOUNDARY COMPUTATION

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

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

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

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

The condition boundary will be

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

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

(x_b, y_1, z_1), (x_b+e, y_1, z_1), and (x_b-e, y_1, z_1). Here, e is a small positive number, e.g., e = (range of x) / 100.

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

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

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

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

4. IMPLEMENTATION AND RESULTS

The framework prototype is designed to process Ada source code and is implemented on a SUN SPARC station using the C language and CLIPS [CLI87], an expert system building tool. Currently, the parser/scanner is used only to generate the code structure file for the test case generator. Since the code instrumentation is a compiler oriented task which is not the major concern of this research, the code is manually instrumented. However, we are now in the process of automating the code instrumentation process using a DIANA interface package [DIA90] for the Verdix Ada Development Systems.

The prototype handles only a Pascal-like subset of Ada. Only subprogram input parameters are considered as input to the unit under test, and input is restricted to integer and float data types. The system has been tested on various Ada programs. We present the results of running the prototype system on two such Ada programs. More results can be found in [BRO90].

Figure 5a shows the first test program graphically pretty-printed with a control structure diagram [CRO89]. Figure 5b shows the condition branching graph that can be abstracted from this control structure diagram. In order to experiment with rule sets reflecting various testing strategies, and to provide a basis for comparing these strategies, rules were grouped into the following categories.

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

B. *Rules that take the best test cases for conditions and generate new test cases by* *incrementing and decrementing the input variables by a percentage.* For results reported here, the rules increment and decrement input variables by 40% of their value rather than a percentage of their range as discussed in section 3.2.1.

C. *Rules that symbolically evaluate values of input variables at condition boundaries and generate new values for input variables that are clustered around these boundaries.* These rules implemented the ideas that were discussed in section 3.2.2. The rule set used in these tests did not take into account conditions in the execution path leading to the target condition and it's associated best test case, but utilized only information about the boundary value of the target condition and generated new test cases as follows.

Given a condition, the best test case for that condition, and an input variable, the value of the variable at the boundary was symbolically determined. Then keeping the other input variables fixed, new test cases were generated by supplying the boundary value of the input variable as input. Additionally, this value was incremented and decremented by a small amount to provide two other test cases. These rules were applied to each input variable appearing in the condition.

For each of the above rule strategies, Figure 6 shows the results of execution of the prototype. The table reports the test cases for which one of the three strategies found new branches in the Ada program. Because all three rule sets were initialized with the same initial test cases, the first test case covered the same path through the code. On the third test case, for example, the table shows that random test case generation rules found another path through the code adding three more branches to the number of branches covered by this rule set. Figure 7 graphically presents the information from the table with the y axis showing what percentage of the 20 branches were covered by the test cases. One should note that the x axis is not a linear scale, and as such the graph does not accurately reflect the relative speed with which maximum coverage is achieved.

Figures 8a and 8b show the source code and the flow graph for the second Ada test program, respectively, and Figure 9 the results of the three testing methods. In this test, complete branch coverage was achieved by the symbolic evaluation rule set in 21 test cases. The increment/decrement rule set found the final branch on the 87th test case, and random test case generation was considerably worse.

In these test programs, the rule-based testing based on heuristic rules, which reflects increment/decrement or symbolic evaluation strategies, outperformed random test case generation. The symbolic evaluation strategy outperformed the increment/decrement strategy in these two examples. But when external functions are called by the test programs (not shown here), the symbolic evaluation strategy appeared to be slightly behind the increment/decrement strategy. The reason is that the symbolic evaluator has no idea what the functions do, so it is unable to successfully determine a branching condition.

5. CONCLUSION

A framework of generating test cases for software branch coverage using heuristic rules has been described. Major framework components include a parser/scanner, a test case generator, a test case analyzer, and a report generator. The parser/scanner instruments a source code and constructs a structure file for the code. The test case generator produces test cases based on heuristic rules and previous coverage and cases. Test case generation always tries to cover some partially covered conditions. The test case analyzer runs new cases in the instrumented code and performs coverage analysis and test case goodness evaluation. The report generator integrates the coverage information and prints test results in table-like forms. By combining coverage analysis techniques, test case goodness evaluation methods, and a rule-based approach, more efficient test case generation can be achieved.

6. REFERENCES

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

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

[BRO90] D.B. Brown, The Development of a Program Analysis Environment for Ada-Phase II (Task I) - NASA Annual Report, Department of Computer Science and Engineering, Auburn University, August 1990.

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

[CRO89] J.H. Cross II, K.I. Morrison, C.H. May, and K.C. Waddel, A Graphically Oriented Specification Language for Automatic Code Generation (phase 1), NASA Annual Report, Department of Computer Science and Engineering, Auburn University, August 1989.

[DEA91] W. H. Deason, D. B. Brown, K.-H. Chang, and J. H. Cross II, "A Rule-based Software Test Data Generator," IEEE Trans. on Knowledge and Data Engineering, March 1991. (To appear)

[DIA90] DIANA Interface Package Manual, Verdix Corp., CA. 1990.

[DUR81] J.W. Duran and S. Ntafos, "A Report on Random Testing," Proc. 5th Int. Conf. on Software Engineering, March 1981.

[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.

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

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

[VOU86] M.A. Vouk, D.F. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software," Proc. of the IEEE Workshop on Software Testing, 1986.

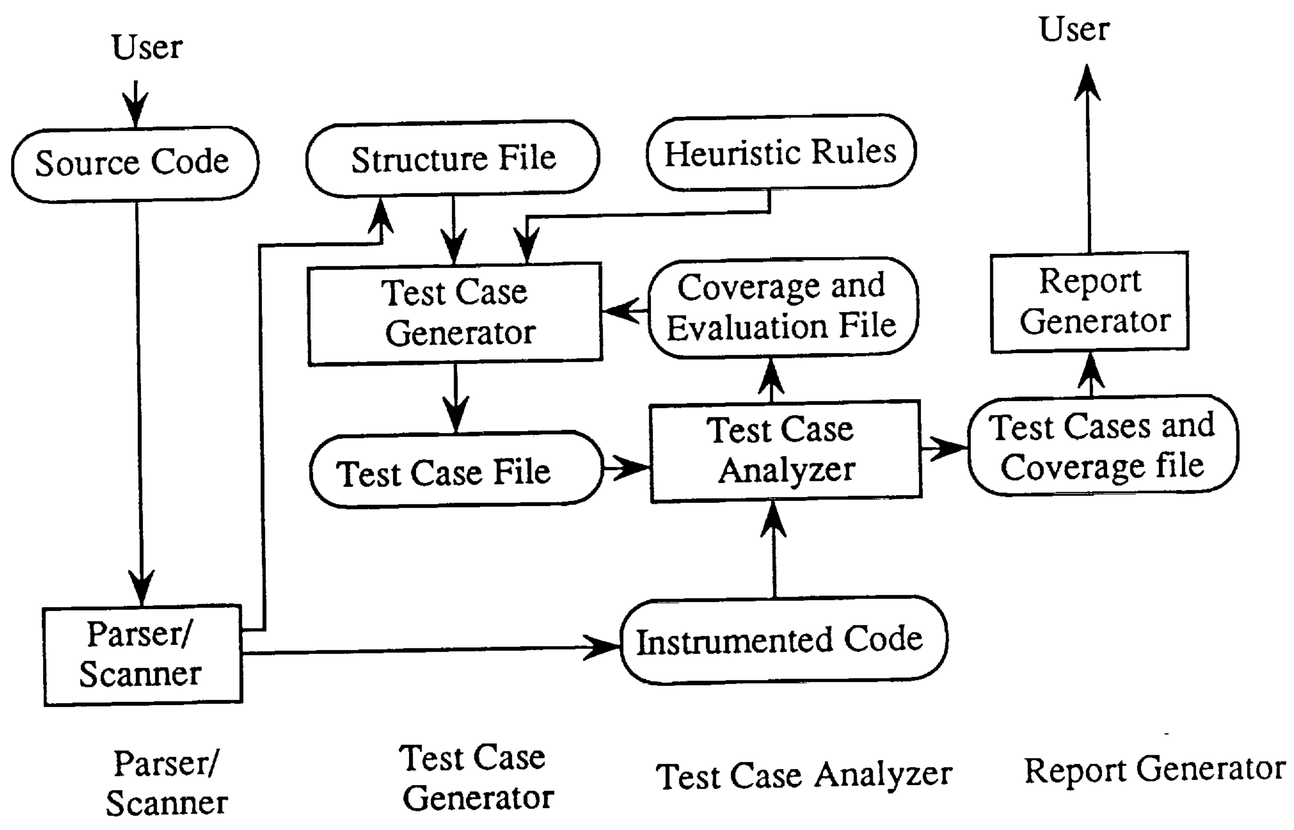


Figure 1 System Framework

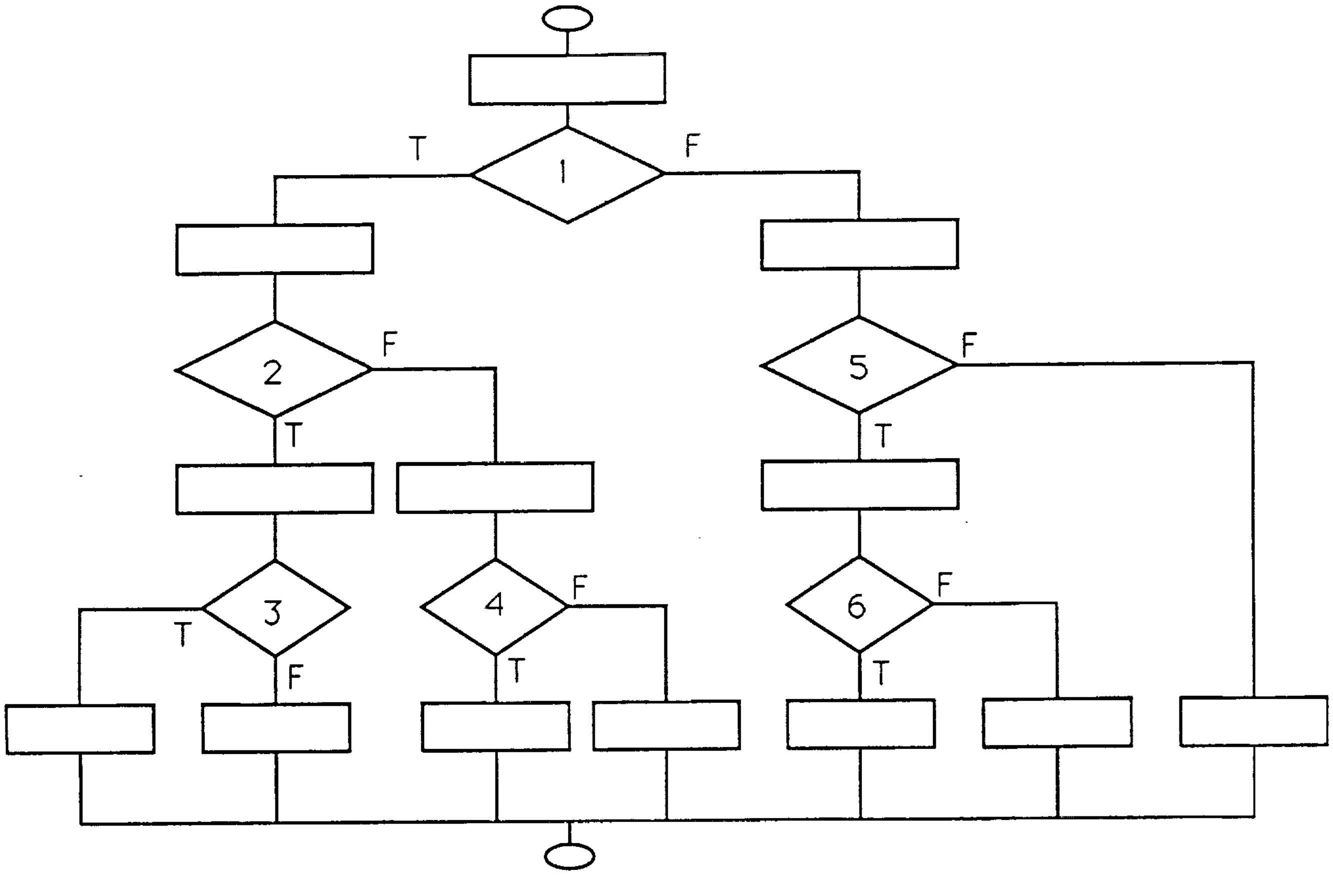


Figure 2a A sample flow chart

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

Figure 2b Coverage table of Figure 2a

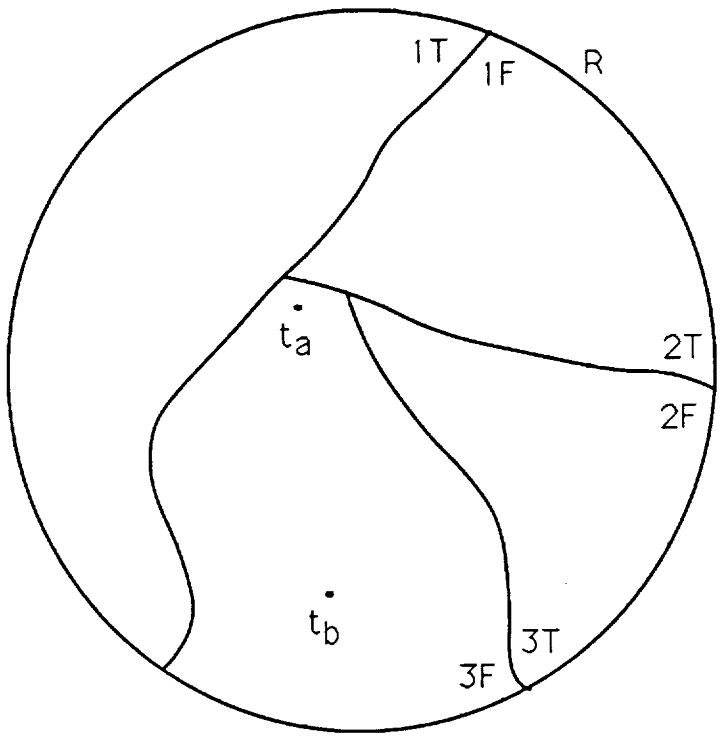


Figure 3 Input space of the program in Figure 2a

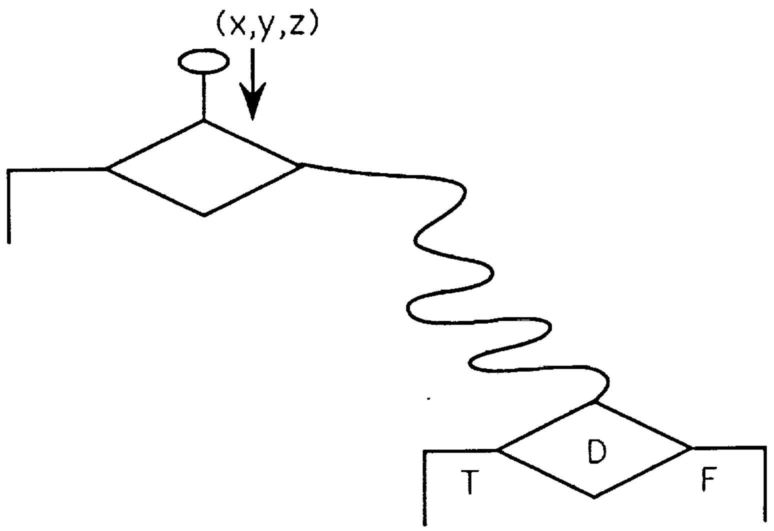
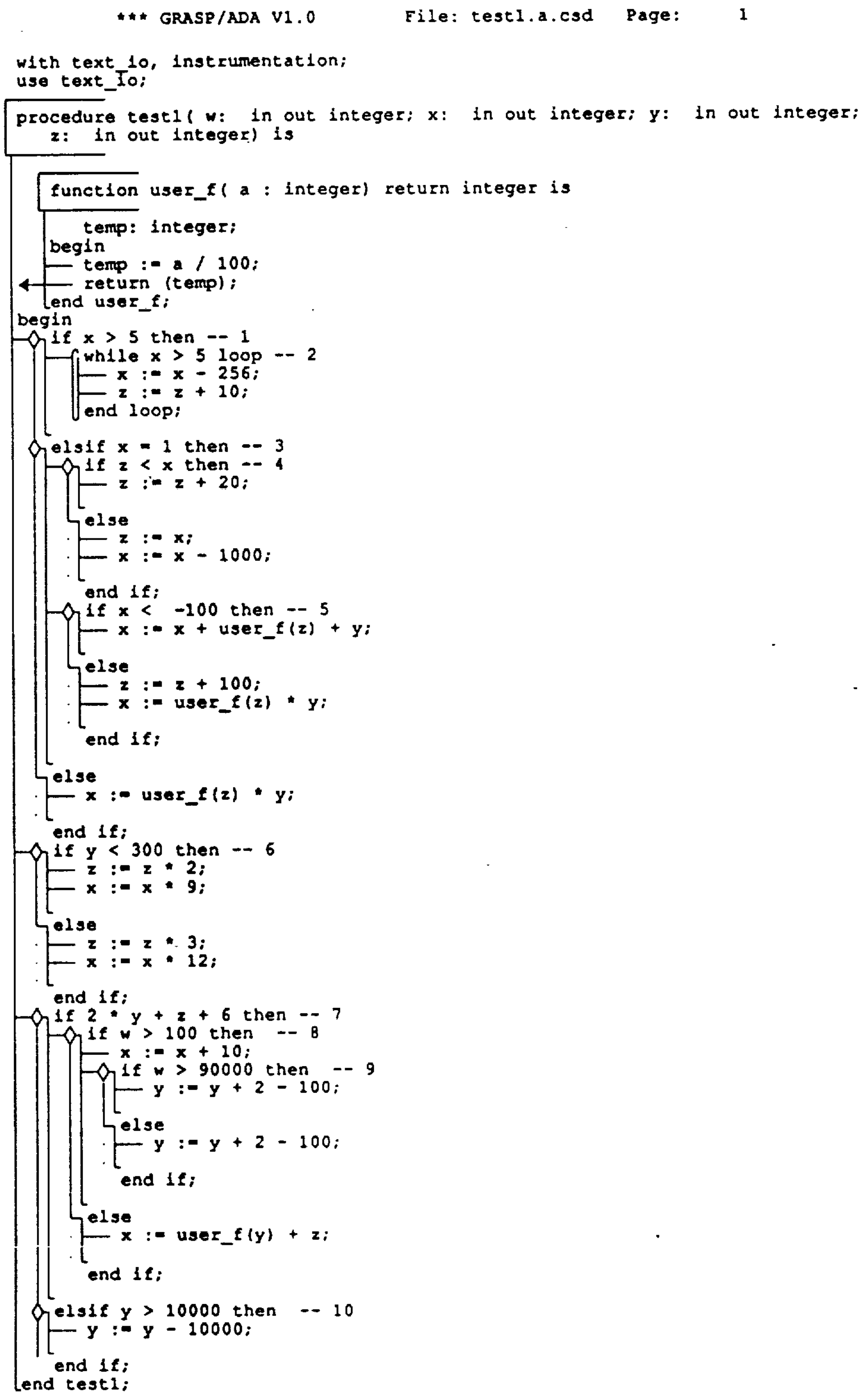


Figure 4 A test case (x, y, z) drives condition D



```
        *** GRASP/ADA V1.0        File: test1.a.csd   Page:      1

with text_io, instrumentation;
use text_io;

procedure test1( w:  in out integer; x:  in out integer; y:  in out integer;
   z:  in out integer) is

   function user_f( a : integer) return integer is

      temp: integer;
   begin
      temp := a / 100;
      return (temp);
   end user_f;
begin
   if x > 5 then -- 1
      while x > 5 loop -- 2
         x := x - 256;
         z := z + 10;
      end loop;

   elsif x = 1 then -- 3
      if z < x then -- 4
         z := z + 20;

      else
         z := x;
         x := x - 1000;

      end if;
      if x <  -100 then -- 5
         x := x + user_f(z) + y;

      else
         z := z + 100;
         x := user_f(z) * y;

      end if;

   else
      x := user_f(z) * y;

   end if;
   if y < 300 then -- 6
      z := z * 2;
      x := x * 9;

   else
      z := z * 3;
      x := x * 12;

   end if;
   if 2 * y + z + 6 then -- 7
      if w > 100 then  -- 8
         x := x + 10;
         if w > 90000 then  -- 9
            y := y + 2 - 100;

         else
            y := y + 2 - 100;

         end if;

      else
         x := user_f(y) + z;

      end if;

   elsif y > 10000 then  -- 10
      y := y - 10000;

   end if;
end test1;
```

Figure 5a   Ada Source Code for Test Program 1

ORIGINAL PAGE IS
OF POOR QUALITY

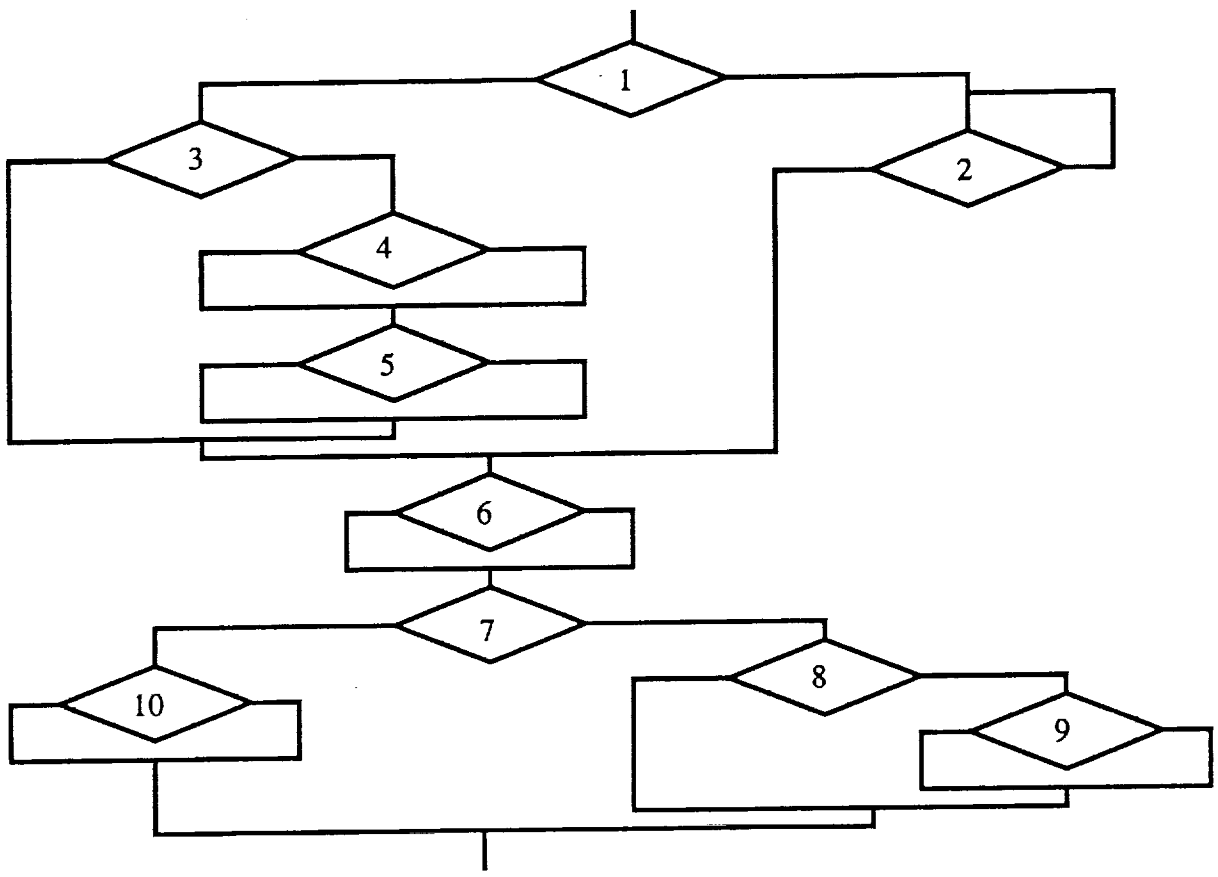


Figure 5b Flow Graph of the First Test Program

Test Case	Inc/Dec by 40%	Random	Symbolic
1,	5,	5,	5
3,	5,	8,	5
7,	5,	8,	8
8,	5,	9,	8
15,	8,	9,	8
21,	8,	10,	8
27,	8,	10,	11
52,	8,	10,	12
56,	8,	10,	13
76,	8,	10,	15
135,	8,	10,	17
232,	8,	10,	19
870,	11,	10,	19
1105,	13,	10,	19

Figure 6 Result for the First Test Program in Table Form

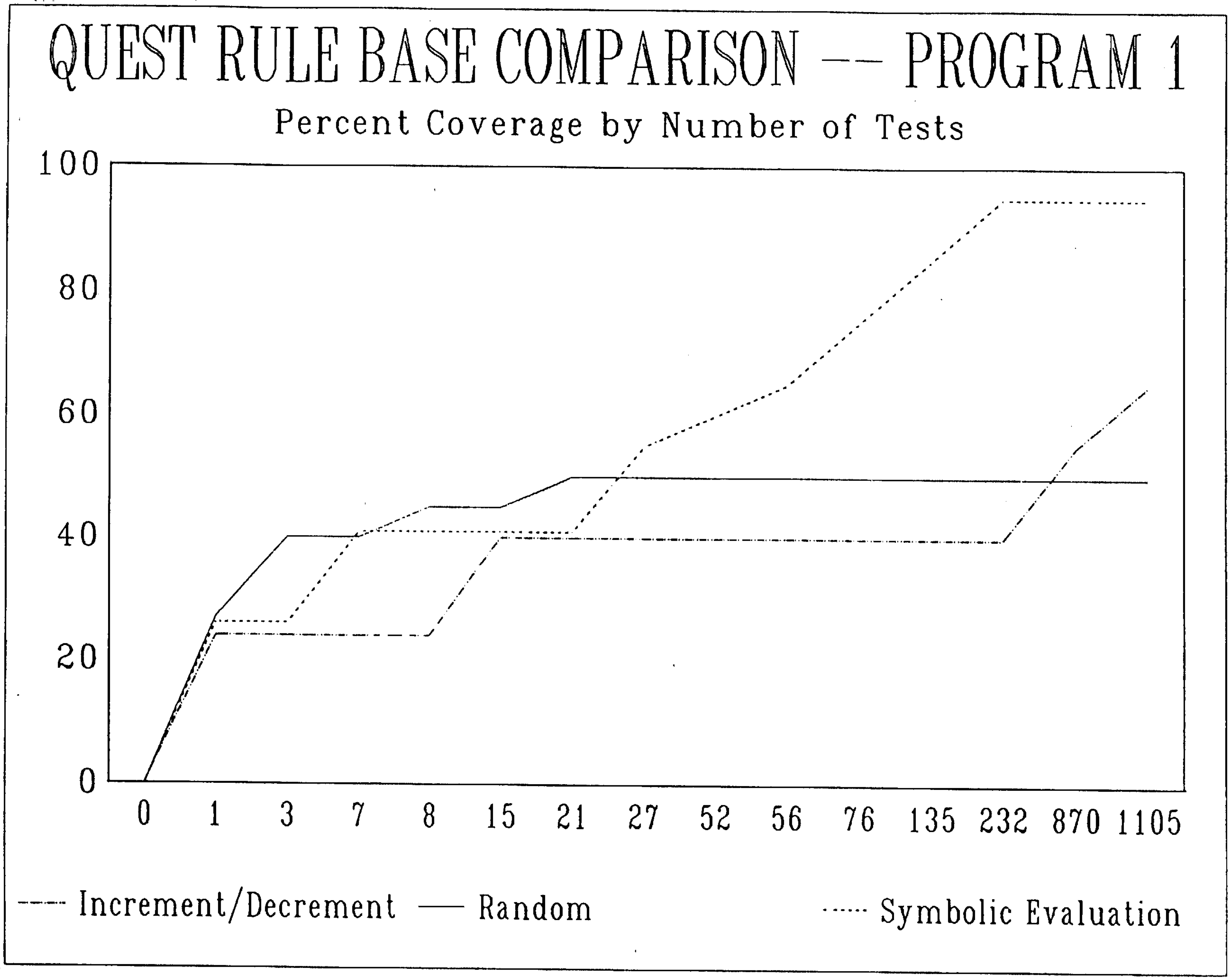


Figure 7 Result for the First Test Program in Graphical Form

```
          *** GRASP/ADA V1.0        File: test2.a.csd   Page:      1

with text_io, instrumentation;
use text_io;
procedure test2( x:  in out integer; y:  in out integer; z:  in out integer)
   is
   t1, t2 : integer;
   function user_f( a : integer) return integer is
      temp: integer;
   begin
      temp := a / 100;
      return (temp);
   end user_f;
begin
   x := x - 100;
   t1 := y - 100;
   t2 := z - 100;
   while x > 1 loop -- 1
      x := z - 2000;
      if y < 10 then -- 2
         if y >  -5 then -- 3
            y := y + t2;
            loop
               z := z +  abs (y);
               t1 := 2 * y;
               if z > 19950 then -- 4
                  y := y + 10;
               else
                  y := y - 10;
               end if;
               z := z - 1000;
               exit when z > 20000; -- 5
            end loop;
         else
            if y >  -20 then -- 6
               z := z - 10000;
            else
               z := z - 20000;
            end if;
         end if;
      end if;
      y := x + z + user_f(t2);
      z := user_f(y) + user_f(t1);
   end loop;
   if y < 100 then -- 7
      y := y + 100;
   end if;
end test2;
```

Figure 8a Ada Source Code for Test Program 2

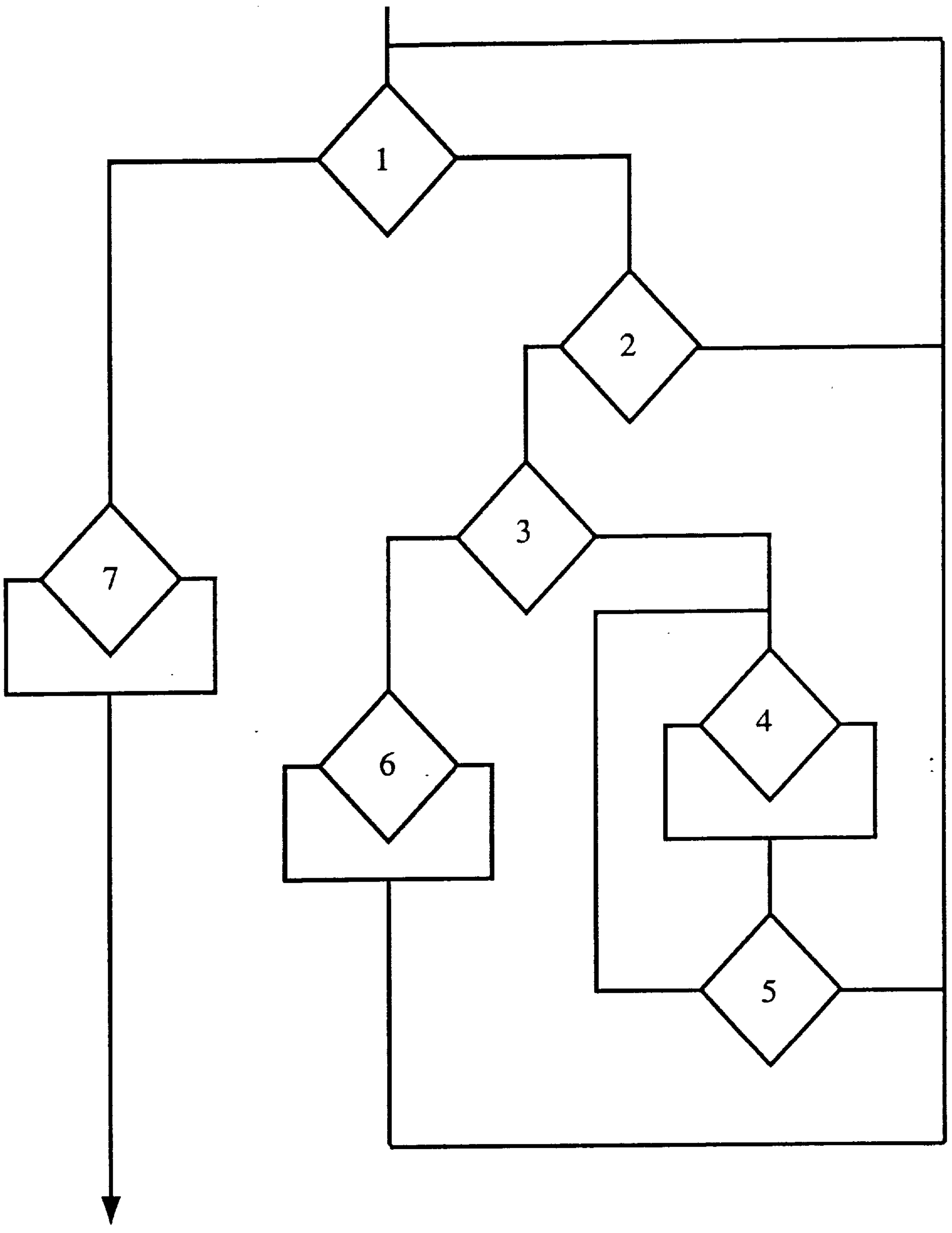


Figure 8b Flow Graph of the Second Test Program

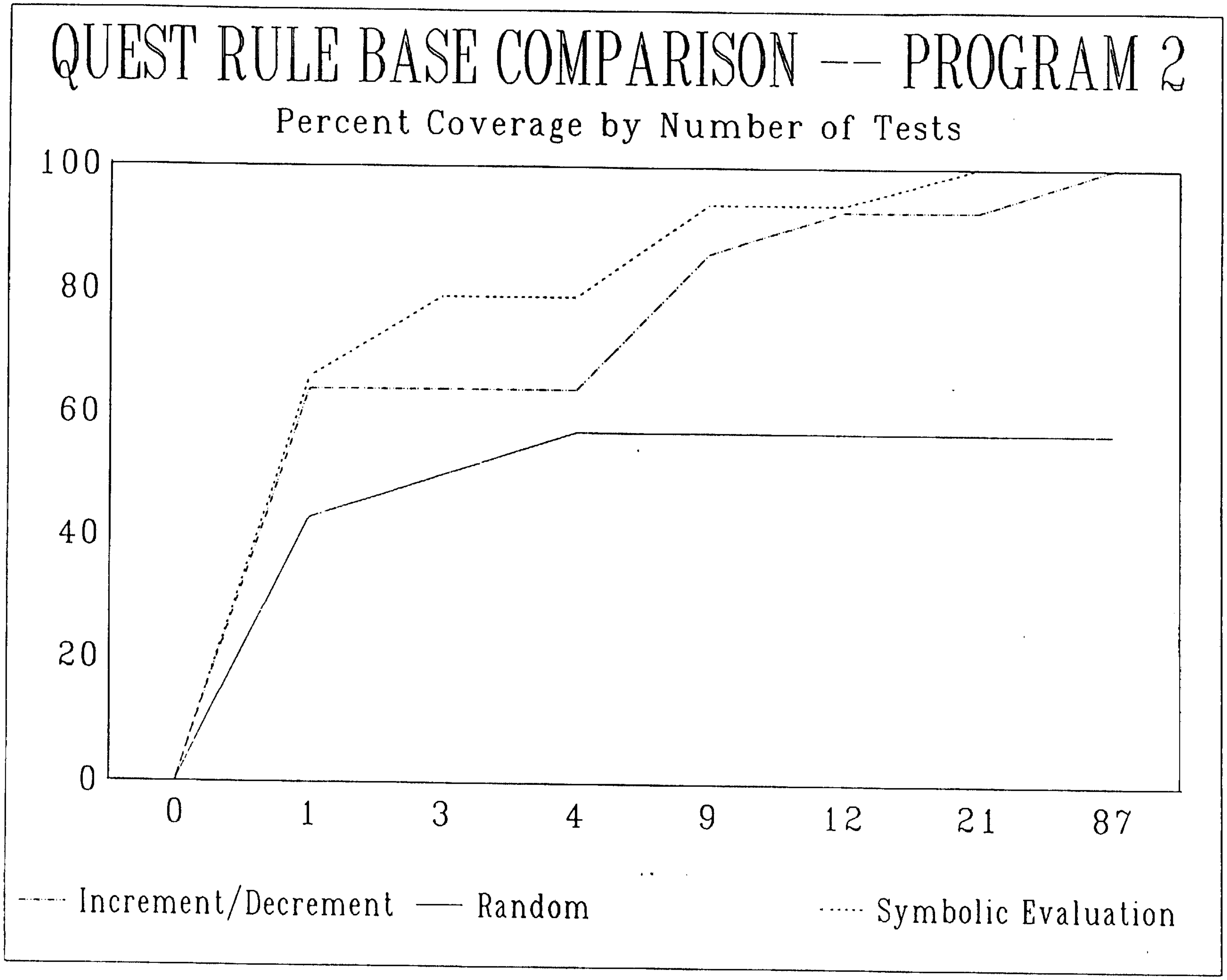


Figure 9 Result for the Second Test Program in Graphical Form

APPENDIX E. PAPER: A FRAMEWORK FOR INTELLIGENT TEST DATA GENERATION BY CHANG, CROSS, CARLISLE AND BROWN SUBMITTED TO THE JOURNAL OF INTELLIGENT AND ROBOTIC SYSTEMS THEORY AND APPLICATIONS (JIRSTA)

A FRAMEWORK FOR INTELLIGENT TEST DATA GENERATION[*]

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

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

ABSTRACT

Test data generation using traditional software testing methods generally requires considerable manual effort and generates only a limited number of test cases before the amount of time expended becomes unacceptably large. A rule-based framework that will automatically generate test data to achieve maximal branch coverage is presented. The design and discovery of rules used to generate meaningful test cases are also described. The rule-based approach allows this framework to be extended to include additional testing requirements and test case generation knowledge.

Keywords: Rule-based Systems, Goodness Values, Software Engineering, Artificial Intelligence, Software Testing, Branch Coverage.

[*] This work is supported in part by George C. Marshall Space Flight Center, NASA/MSFC, AL 35812 (NASA-NCC8-14).

1. INTRODUCTION

Program testing methods can be classified as either dynamic analysis, static analysis, or a combination of these [RAM75]. Dynamic analysis of a program involves executing the program with test cases and analyzing the output for correctness, while static analysis includes such techniques as program graph analysis and symbolic evaluation. A dynamic test strategy is a method of choosing test data from the functional domain of a program. It is based on criteria that may reflect the functional description of a program, the program's internal structure, or a combination of both [ADR82]. These criteria specify the method of test case generation to be used for a dynamic test strategy.

Generating test inputs to a program may not appear to be a difficult problem since it may be done by a random number generator [DUR81]. However, random testing alone has been shown to be an inadequate method for exposing errors. When combined with extremal and special value (ESV) testing, it can be an effective method and can provide a direction for the generation of future test cases [VOU86]. On the other hand, algorithms for generating test data to satisfy particular adequacy criteria have generally poor time and space complexities and produce small amount of test data.

The objective of this research is to design and develop a framework that will automatically generate test data to achieve maximal branch coverage of an arbitrary program. Also included is the design and discovery of rules that can be used to generate meaningful test cases. A rule-based approach allows this framework to be extended to include more testing requirements and test case generation knowledge.

2. FRAMEWORK

The outline of the framework is shown in Figure 1. It is divided into four major components: parser/scanner, test case generator, test case analyzer, and report generator. For simplicity, examples have been restricted to IF-THEN and IF-THEN-ELSE statements with conditions of single expressions.

2.1 PARSER/SCANNER

The purpose of the parser/scanner is to instrument the input source code and create a file containing information about the input code's structure. The instrumented code is used by the test case analyzer to analyze the coverage of the code. The structure file provides the test case generator information about the code so that test cases can be generated accordingly.

The instrumentation of the code is done by inserting a function call into the condition part of an IF-THEN statement. For example, statement IF (8*Y-4*X+5) >= (5-Z) THEN do-1 ELSE do-2 will be instrumented as IF analyze((8*Y-4*X+5) >= (5-Z)) THEN do-1 ELSE do-2. "Analyze" is a function defined in the test case analyzer. It calculates and returns the truth value of the condition to the statement. This keeps the performance of the original code unchanged. However, in this process "analyze" also performs coverage analysis and other evaluation tasks (see the section of test case analyzer).

The structure file contains information about input parameters and conditions/branches of the code. For each input parameter, its data type and range are recorded. An IF-THEN statement is recognized as IF LHS <op> RHS THEN do-1 ELSE do-2, where RHS and LHS stand for right-hand-side and left-hand-side of the condition expression respectively. "<op>" denotes a logical operator such as <, >, =, =<, >=, or <>. In addition to RHS, LHS, and <op>, information about parameters that appear in the LHS and RHS is also included in the structure file. This information is represented as facts so a knowledge-based test case generator can access it easily.

2.2 TEST DATA GENERATOR

The test data generator produces new test cases that are intended to cover the branches associated with conditions in the code. It takes coverage results from the analyzer and code information from the structure file and determines what conditions/branches should be targeted for new case generation. It then uses test case generation rules to generate more test cases. When a set of new cases is generated, it is stored in a test case file. The test case analyzer will then run these test cases in the instrumented code and record the cumulative coverage of conditions and branches. The coverage and other evaluation results are then fed back to the test case generator for further case generation. The test case generation and analysis cycle is repeated until a predefined coverage is reached or a certain number of test cases has been generated. In addition to this regular test case generation, a test case generator also may be used to generate initial cases as indicated in Figure 2. If the user provides initial cases, this step can be skipped and the test case analysis can be started immediately; otherwise, the test case generator has to generate initial cases to start the cycle. They can be generated by either random number generation methods or simple heuristics. One heuristic is to assign each input parameter to its mid-range, lower-bound, and upper-bound. This heuristic would give three initial cases. For example, if the input parameters are (x y) and the ranges for x and y are [0 100] and [-10 0] respectively, the three cases will be (50 -5), (0, -10), and (100, 0).

2.3 TEST CASE ANALYZER

The test case analyzer runs the new test cases in the instrumented code, records the branch coverage, and evaluates the "goodness" of each test case. The analysis result is then fed back to the test case generator for further case generation. When the test case generation and analysis cycle is terminated, an analysis report is generated.

The coverage recording task is a simple table filling process which keeps track of what conditions have been fully, partially, or not yet been covered. The test case generator uses this information to select a target branch to generate more cases. A "goodness" value is used to select a test case from a set of test cases that drives the code in a particular way. This selected test case will be used as a model for further case generation. Detailed discussion is given in a later part of the paper.

2.4 REPORT GENERATOR

When the test case generation and analysis cycle is completed, the report generator will print the results for the user. The information to be included in the report can be specified by the user. These include test cases, condition and branch coverage, and statistics of the cases and coverage.

The report generator and the parser/scanner involve mechanic types of processings. Although the effort required to implement these two components is great, technically they are well understood. Detailed discussion of these two components can be found in [BRO89]. The remainder of this paper will provide a detailed discussion of the methodologies used for test data generation and analysis.

3. TEST DATA GENERATION STRATEGY

The objective of the Test Data Generation (TDG) component of the testing methodology is to achieve maximal branch coverage. In order to ensure the direction of test case generation is 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 3a and 3b 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 3a and 3b. Currently, conditions 1 and 2 are fully covered; conditions 3, 4, and 5 are partially covered; and condition 6 is not covered. Since conditions 1 and 2 are fully covered, there is no need to generate more cases 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 3b, will not be targeted for new case generation. This is because no test case model yet exists that can be used for modification. A test case model will eventually surface later in the process, and in this example, after condition 5 is fully covered, a model for condition 6 will appear.

3.1 BEST TEST CASES

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 3b, 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 is 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-THEN statement: IF exp THEN do-1 ELSE do-2. The evaluated Boolean value of exp determines the branching. Exp can be expressed in the form of: LHS <op> RHS. 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

$$| \mathrm{LHS}\ (t1) - \mathrm{RHS}\ (t1)\ | \ /\ \mathrm{MAX}\ (\ | \mathrm{LHS}\ (t1) |, | \mathrm{RHS}\ (t1) |\ ) \qquad (1)$$

LHS(t1) and RHS (t1) represent the evaluated value of LHS and RHS, respectively, when t1 is used as the input data. This measure tells the closeness between LHS and RHS [DEA90]. 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 generally 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 slightness of modification of a test case as its goodness, this measure is computed based on the target condition only. A slight modification to the LHS and RHS of the target condition may not have the same meaning to those conditions on the path. This may result in what we will call unanticipated branchings along the path, that is, a flow of control that may no longer reach the condition under test. In order to reduce the likelihood of unanticipated branching, a test case's goodness measure should also consider those conditions that are on the path leading to the condition. This idea can be expressed in the following example.

In Figure 4, 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 4). 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.

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:

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

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 in equation (2), and P(t,D) is defined as:

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

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 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.

For example in Figure 5. Input to the procedure contains three parameters x, y, and z. Assume condition D is partially covered, its best test case is (x_1, y_1, z_1), and we need to generate more cases to cover D's false branch. Condition D can be expressed as LHS($x, y, z, v_1, v_2, \ldots$) <op> RHS($x, y, z, v_1, v_2, \ldots$). Here, $v_1, v_2, \ldots$ 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 test case (x_1, y_1, z_1) is input into the procedure, the evaluation of D will result in a truth value. The following sections discuss some heuristics that can be used to generate new cases.

3.2.1 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 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.

3.2.3 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, developed by DeMillo, Lipton, and Sayward [DEM78], and Howden [HOW87], were intended originally to be applied in manual test case generation. Implementation of this kind of heuristic has been reported in a separate paper [DEA90], in which these rules are represented in Prolog. Performance of this approach shows a significant improvement over randomly generated test cases.

3.2.4 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,

$$x_b = f1\ (y,z,v_1,v_2, \ldots)$$
$$y_b = f2\ (x,z,v_1,v_2, \ldots)$$
$$z_b = f3\ (x,y,v_1,v_2, \ldots)$$

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

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

The condition boundary will be

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

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., $x_b = f1\ (y,z,v_1,v_2, \ldots)$. In order to compute the desired value of x at D, use the actual values of y, z, v_1, v_2, ... just before D is evaluated. The computation provides the desired boundary value of x at condition D. Three new cases can be generated to cover both 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 = (range of x) / 100. Similarly, this case generation procedure can be applied to variables y and z.

In this procedure, 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      $x_1 = x_c$
THENgenerate 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 $x_i <> x_c$. 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:

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

Formula (5) is identical to (2) but has a different interpretation. Following this measurement, we can modify input x based on this percentage. One more question needs to be answered: how should the amount of modification 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., [upper_limit_of_x - 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 vary x by one percent of x's range, i.e., e = 12, six new cases can be generated. In this example, 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      $x_l = x_c$
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 (6),
        generate new cases for being on or slightly off boundary.
```

$$x_i \pm |x_b - x_c| / (2*\mathrm{MAX}(|x_b|, |x_c|)) * (x_{largest} - x_{smallest}) \qquad (6)$$

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.

4. IMPLEMENTATION

The current prototype testing system is designed to process Ada source code and is implemented on a VAX-780 using the C language and CLIPS [CLI87], an expert system building tool which is written in C. Currently, the parser/scanner is used only to generate the code structure file for the test case generator. Since the code instrumentation is a compiler oriented task which is not the major concern of this research, the code is manually instrumented. However, we are now in the process of automating the code instrumentation process. The code information generated by the parser/scanner is stored as CLIPS facts in a file. Examples of facts are shown below.

```
(varl 2 x y)                          ;variable list
(exp 1 l x "+" y)                     ;LHS expression of condition 1
(exp 1 r 2 "*" x "*" y "-" 10)        ;RHS expression of condition 1
(v 1 I 1 100)                         ;type and range of 1st variable
(v 2 I -5 20                          ;type and range of 2nd variable
```

The test case generator is written in CLIPS which uses production rules. CLIPS was selected because it provides a fully integrated environment for C. Rules for the fixed percentage and the random modifications have been implemented. Boundary and symbolic computation rules are being added to the system. As an example, a fixed percentage modification rule is shown below.

```
;this rule uses the best test case associated with a condition,
;and for n input variables, generate n test cases by altering
;each variable one percent of its range.

(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 test case analyzer, written in C, runs each test case from the test case file, evaluates goodness values for each case, and records the coverage. It is designed in a way that the definition of test case goodness can be selected by the user.

The report generator, written in C, provides a listing of test cases and statistics. An example of a statistical printout from the prototype is shown below.

```
****************************************************************

             QUEST CUMULATIVE COVERAGE REPORT

-- # of decision:          8    ;total condition number
-- # of decision fanouts:  16   ;total branch number
-- # of fanouts hit:       15   ;branches hit
-- Decision coverage:      93%
-- Decisions not hit:      4T   ;decisions not hit
-- Condition not hit:           ;conditions not hit

****************************************************************

-- Decision:            1
-- Evaluated true:      1165    ;number of times 1T is hit
-- Evaluated false:     155     ;number of times 1F is hit

****************************************************************
```

Current effort of the framework development is to test more programs and collect performance statistics. Form the statistics, performances of various goodness evaluations and heuristic rules will be compared to determine their effectiveness.

5. CONCLUSION

A framework of generating test cases for software branch coverage using heuristic rules has been described. Major framework components include a parser/scanner, a test case generator, a test case analyzer, and a report generator. The parser/scanner instruments a source code and constructs a structure file for the code. The test case generator produces test cases based on heuristic rules and previous coverage and cases. Test case generation always tries to cover some partially covered conditions. The test case analyzer runs new cases in the instrumented code and performs coverage analysis and test case goodness evaluation. The report generator integrates the coverage information and prints test results in table-like forms. By combining coverage analysis techniques, test case goodness evaluation methods, and rule-based approach, more efficient test case generation can be achieved.

The contributions of this framework include the following: (1) an approach to generate test cases from previous cases, (2) methods of evaluating test cases with respect to a condition, (3) some heuristics for test case generation, and (4) an extensible framework, i.e., more evaluation and heuristics can be added easily. We are currently running more test programs. Coverage statistics of the test runs will provide an in depth comparison with other conventional test generation methods.

6. REFERENCES

[ADR82] W.R. Adrion, et at., Validation, Verification, and Testing of Computer Software, ACM Computing Surveys, Vol. 14, June 1982.

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

[BRO89] D.B. Brown, The Development of a Program Analysis Environment for Ada-Phase II (Task I) - NASA Six-month-report, Department of Computer Science and Engineering, Auburn University, December 1989.

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

[DEA90] W. H. Deason, D. B. Brown, K.-H. Chang, and J. H. Cross II, "A Rule-based Software Test Data Generator," IEEE Trans. on Knowledge and Data Engineering, 1990. (To appear)

[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.

[DUR81] J.W. Duran and S. Ntafos, A Report on Random Testing, Proc. 5th Int. Conf. on Software Engineering, March 1981.

[HOW86] W.E. Howden, "A Functional Approach to Program Testing and Analysis," IEEE Trans. on Software Engineering, Vol. SE-12, No. 10, October 1986.

[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.

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

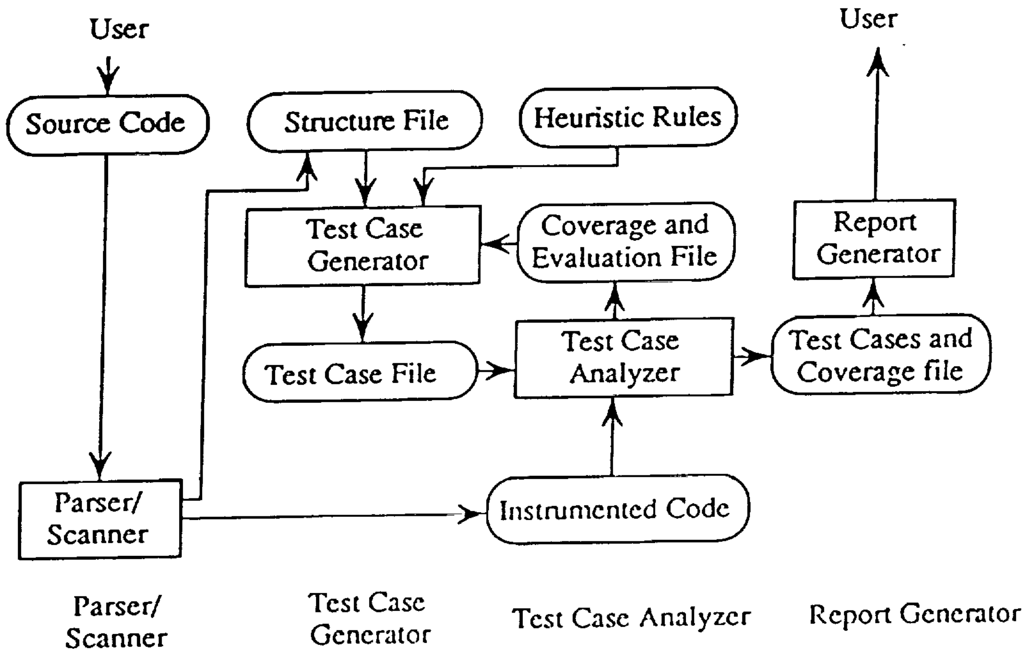


Figure 1 System Framework

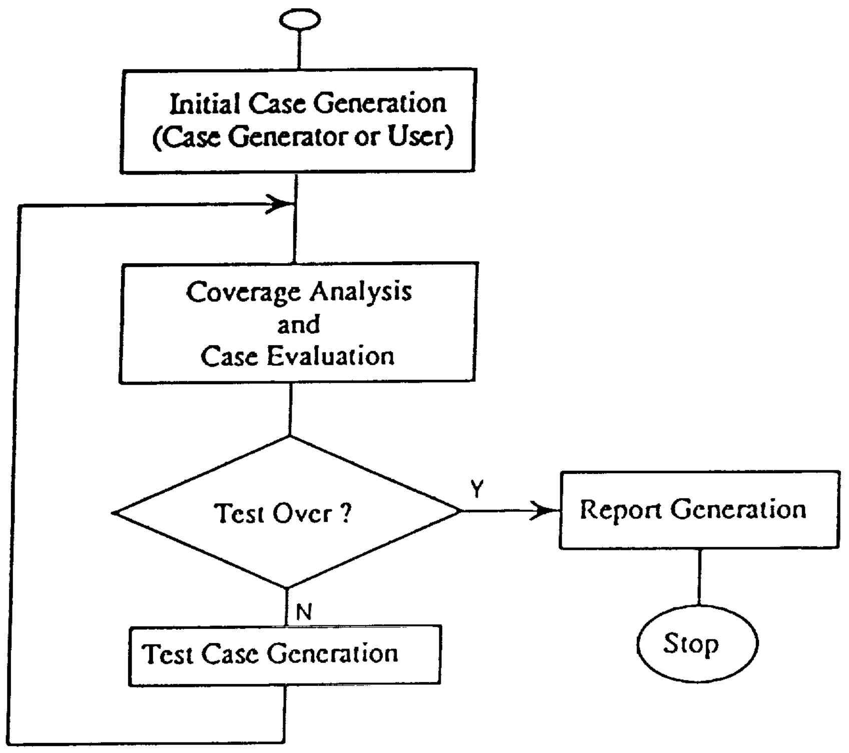

Initial Case Generation (Case Generator or User)
Coverage Analysis and Case Evaluation
Test Over ?
Y
Report Generation
N
Test Case Generation
Stop

Figure 2 Test case generation and analysis cycle

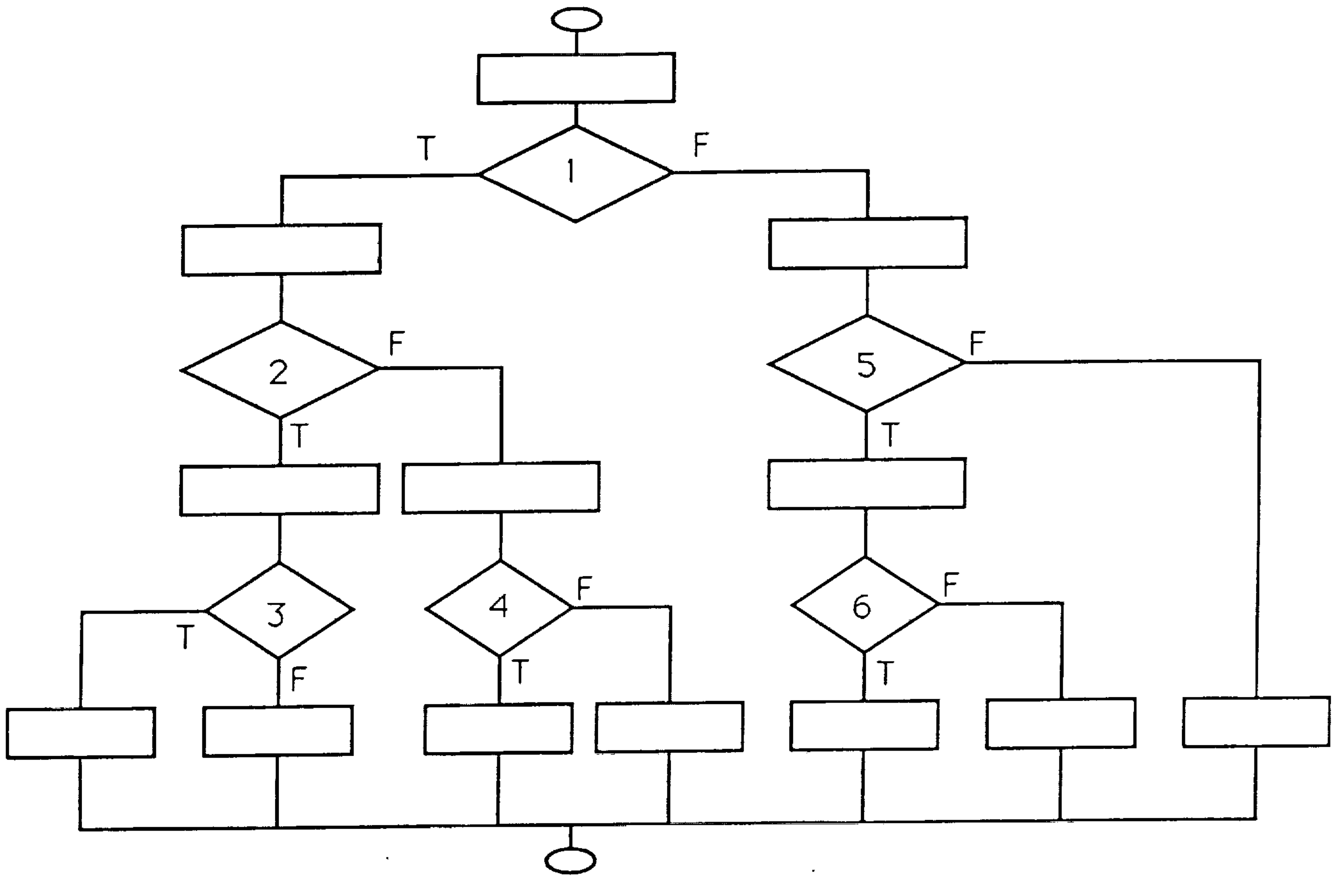


Figure 3a A sample flow chart

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

Figure 3b Coverage table of Figure 3a

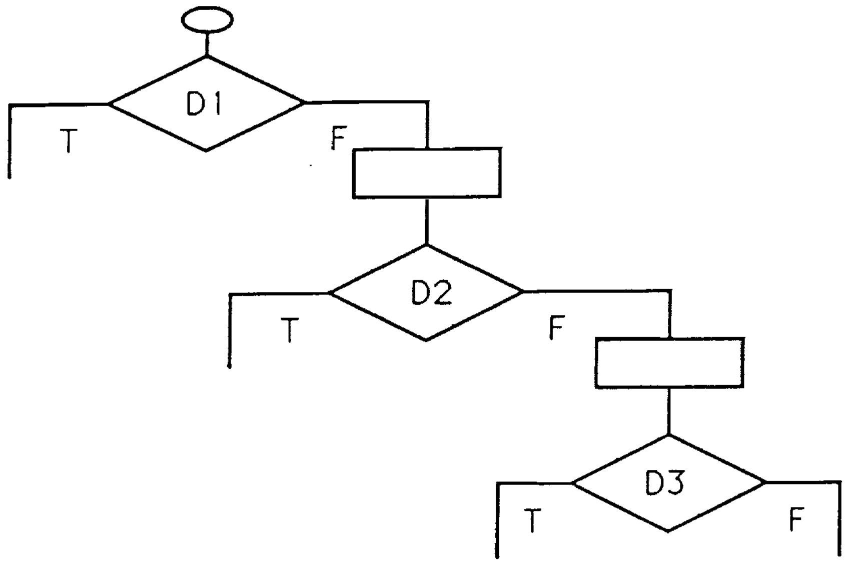


Figure 4a A sample program

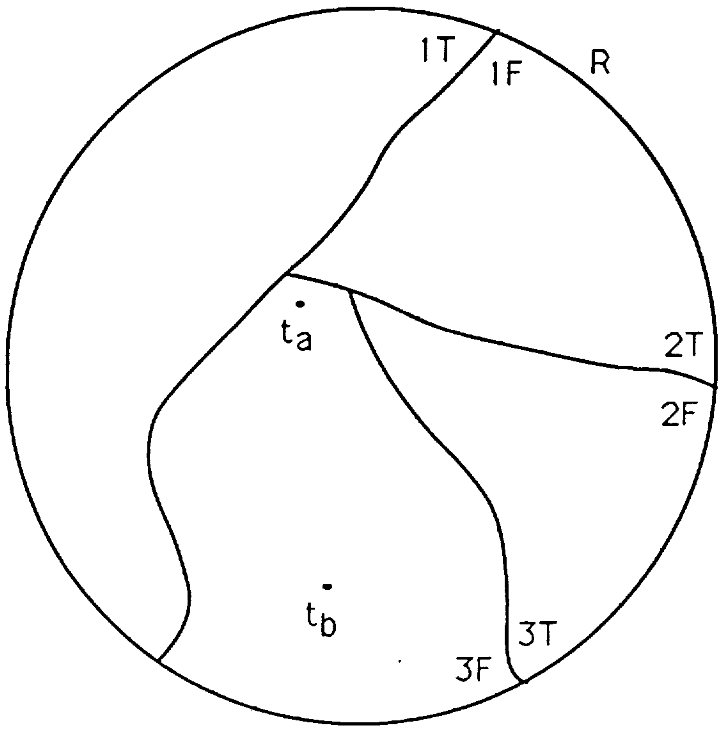


Figure 4b Input space of the program in Figure 4a

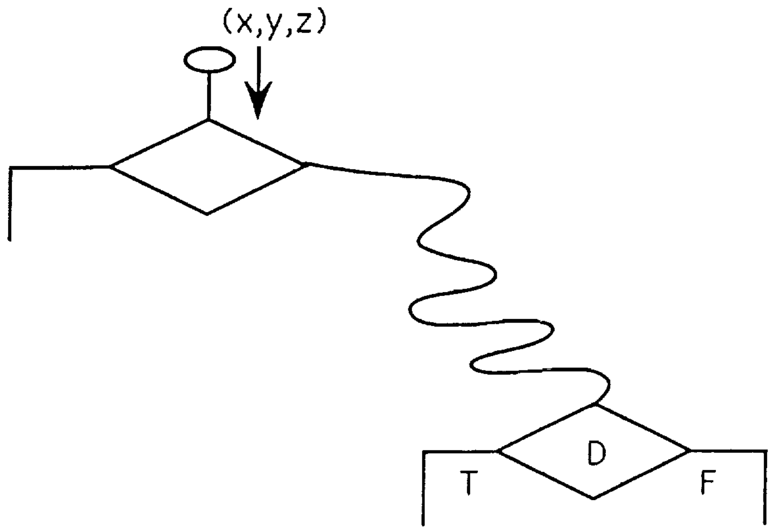


Figure 5 A test case (x, y, z) drives condition D