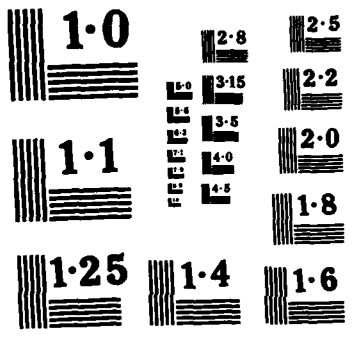
AD-A158 099 PRACTICAL GUIDELINES FOR TESTING ADA PROGRAMS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J B SNAUFER MAY 85 AFIT/CI/NR-85-58T 1/2

UNCLASSIFIED F/G 9/2 NL


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

1

PRACTICAL GUIDELINES FOR TESTING ADA PROGRAMS

by

Joseph Blair Snaufer

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 1985

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-58T — 2. GOVT ACCESSION NO. AD-A158 099	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Practical Guidelines for Testing ADA Programs	5. TYPE OF REPORT & PERIOD COVERED THESIS/~~DISSERTATION~~
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joseph Blair Snaufer	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Arizona State University	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433	12. REPORT DATE May 1985
	13. NUMBER OF PAGES 119
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

[illegible]
LYNN E. WOLAVER
Dean for Research and
Professional Development
AFIT, Wright-Patterson AFB OH

5 AUG 1985

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DDC QUALITY INSPECTED 1

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

85 8 13 108

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: Practical Guidelines for Testing ADA Programs

AUTHOR: Joseph Blair Snaufer

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS ________ ( ) b. $________

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME	GRADE	POSITION
ORGANIZATION	LOCATION	

STATEMENT(s):

PRACTICAL GUIDELINES FOR TESTING ADA PROGRAMS

by

Joseph Blair Snaufer

has been approved

May 1985

APPROVED:

James S. Collofello, Chairperson

[illegible]

Terry Mellon

Supervisory Committee

ACCEPTED:

[illegible]

Department Chairperson

[illegible]

Dean, Graduate College

Author .... Joseph B. Staufer

Title .... Practical Guidelines for Testing Ada Programs

Military Rank ..... Captain

Service Branch .... Air Force

Date 1985

Number of Pages ... 119

Degree Awarded .... Master of Science (Computer Science)

Institution .... Arizona State University

Sources .... IEEE Transactions on Software Engineering
                     Software Engineering with Ada
                     Software Testing Techniques
                     Proceedings of COMPSAC
                     Software System Testing and Quality Assurance
                     Software Engineering: A Practitioner's Approach
                     Proceedings of COMPCON

ABSTRACT

Practical testing guidelines are given for Ada programs. This involves a general testing strategy supplemented with testing guidelines for specialized features. A general background of testing is given which covers testing objectives, testing steps, and current testing approaches for Pascal-like languages. Testing approaches for Pascal-like languages are given because Pascal is the base language for Ada. Ada is then analyzed from a testing point of view. Features are identified that are different from Pascal-like languages. The analysis includes a brief description of each feature and a description of errors/problems that might occur when using the feature. The analysis determines whether Pascal-like language structural testing approaches are adequate for the structural testing of each feature. If a new structural testing approach is needed for a feature, then testing guidelines are presented for that feature. A general testing strategy is then presented. The general testing strategy applies to Ada as a whole. The additional structural testing guidelines for certain features are to supplement the general testing strategy. Case studies are then given which demonstrate the general testing strategy and the supplemental guidelines. Each case study consists of a sample routine, sample test cases, and a step by step description of how to apply the supplemental guidelines to the sample routine. Finally, topics are given for future research.

ACKNOWLEDGEMENT

I would like to thank Dr. James S. Collofello who conceptualized this thesis and who provided valuable direction and insight throughout its development. Thanks also to Dr. Terry Mellon, Dr. Esen Ozkarahan, and Dr. Ernest Hirata for their reading of this document. Finally, I would like to thank my wife, Lisa Snaufer, who provided moral support and patience.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Testing plays an important and costly role in software development. It is important because errors need to be uncovered and there needs to be confidence in the correctness of the software when testing is completed. It is costly and can consume as much as 50% to 80% of the development budget during the software development life cycle [1]. This necessitates the need for thorough, cost-efficient testing methodologies/techniques. Considerable research and development is currently being done in this area [2,3]. A relatively new programming language needing testing research is Ada†.

Ada is a general purpose programming language developed at the initiative of the United States Department of Defense. Ada, which was designed using Pascal as the base language [4], incorporates many features different from the Pascal language. It was designed for the domain of large, real-time, embedded computer systems, based on a set of specific language requirements. As a result of the language requirements, Ada supports

†Ada is a registered trademark of the United States Government (Ada Joint Program Office).

testing. The methodology is based on definition/usage graphs, which focus on the occurrences of variables. The authors justify the use of this strategy by stating that one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

Ideally, the tester can guarantee absence of errors by exhaustive functional testing, but this is virtually impossible to do in large programs. For example, cause-effect testing a large program would require a potentially large, unmanageable number of test cases.

The inability to do exhaustive functional testing is compensated by performing structural testing. Structural testing is defined as constructing and applying tests that cause the execution of certain components of a program [12]. This is done by creating test cases that depend on the internal structure of the program. The actual source code is used to derive test cases. Beizer [1] calls this path testing and contends that it is the cornerstone of all testing. He further states that it is the minimum, mandatory foundation of any effective test plan. A path through a program is any executable sequence of statements through that program that starts at a junction (point in program where control flow merges) or decision and ends at another, or possibly the same, junction or decision [17].

DISCOUNT	QUANTITY
0%	0 < Qty <= 10
5%	10 < Qty <= 50
10%	50 < Qty <= 5000

Figure 1. Classes for Equivalence Class Testing.

Cause-effect testing is similar to equivalence class testing, but it goes two steps further. All possible combinations of equivalence classes are tested and the test cases include the expected results. Because all combinations of equivalence classes are tested, it may be necessary to test an equivalence class more than once. Cause-effect testing provides a representation of logical conditions and corresponding actions [9]. It is also beneficial in that it points out incompleteness and ambiguities in the specifications [15]. The matrix used for equivalence class testing is expanded for cause-effect testing to allow for the expected results, and is now called a decision table.

A method put forth by Rapps and Weyuker [16] is called data flow analysis. The strategy is to track input variables through a program, following them as they are modified, until they are ultimately used to produce output values. By doing this, associations between the definition of a variable and its uses are examined during is 99.99, then try to print 99.99 and 100.00. Again, all of these boundary areas are highly suspect to error.

Equivalence class testing strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases needed [9]. Test cases are derived by partitioning inputs into equivalence classes. An equivalence class represents a set of valid or invalid states for input conditions. Figure 1 gives an example of equivalence class testing by showing discounts a customer would receive when ordering goods. The tester would select one value from each category. From Figure 1, the values 5, 25, and 75 might constitute test cases for each valid equivalence class. Invalid classes also need to be tested. In this example, invalid orders occur when a customer orders less than one good or more than 5000 goods. In this case the values -1 and 6000 could be tested. Test cases are derived so that each equivalence class is tested only one time. To keep track of the testing effort, a matrix should be used which is comprised of input information only. The matrix lists the test case along with the corresponding equivalence class(es) being tested.

Test cases are derived based upon the software's specifications. These test cases can be prepared as specifications are identified resulting in a comprehensive test plan. Preparing test cases early is useful in preventing errors from occurring in the first place. The test plan may be used later to validate the system and is especially useful for user acceptance testing. There are several systematic methods for selecting test case data.

Domain (boundary) testing attempts to uncover errors in a path domain by selecting test data on or near the boundary of the path domain [13]. Many errors occur around a boundary. A path corresponds to some possible flow of control and its domain causes its execution. The primary error that domain testing attempts to uncover is the control flow error. A control flow error occurs when a specific input follows the wrong path due to an error in the control flow of the program [14]. The goal of domain testing is to demonstrate that the boundary of a control flow is correct within an acceptable error bound.

Along with domain testing are other forms of boundary testing. Whereas domain testing tests flow of control (paths) boundaries, one can also test the boundaries of input and output ranges, indices, etc. For example, if the range of some acceptable input is 1-100, then test inputs 0, 1, 100, and 101. If a maximum printable value neither passing tests nor demonstrating compliance imply correctness. If the entire set of test cases have been run and no errors are found, it does not mean that there are no more errors in the program. Perhaps the set of test cases is incomplete.

2.6. Testing Methods

Testing methods are roughly broken down into two categories: functional and structural. The reason for the different categories is that software can be viewed from different perspectives. No one strategy is best, as neither can guarantee finding all errors in a program. The idea is to use different strategies which result in maximizing the number of errors found.

In functional testing, functions are identified which are supposed to be implemented, and the software is tested against the specifications for those functions [12]. Functional testing is conducted to determine the extent to which software conforms to its requirements [2]. This is also called black-box testing. The manner in which specifications are met is ignored. The tester is only concerned with demonstrating that the input is properly accepted, output is correctly produced, and the integrity of data is maintained in accordance with specifications [9].

The builds and threads method involves testing the system by functionality. The system is divided into builds (functions), which are comprised of one or more units. The entire build is tested at the same time. The idea is to bring the system up one function at a time.

Validation provides final assurance that software meets all functional and performance requirements [9]. This is performed by the software engineer utilizing a test plan which outlines the testing procedure. If the software passes this phase then it is ready for release.

System testing involves combining the software with the actual environment it is designed to execute in. The final step of this phase is acceptance testing which involves the user. If the user accepts the software at this point then there is a working system. Note that some users will accept software as is from the vendor based on prior success or the vendor's validation test.

2.5. Fundamental Problem of Software Testing

According to Goodenough and Gerhart [10] the fundamental problem of testing is "the inference from the success of one set of test data that others will also succeed, and that the success of one test data set is equivalent to successfully completing an exhaustive test of a program's input domain." Foster [11] notes that

The second step is integration testing which involves putting together all related units to find errors associated with interfacing. The objective is to take unit-tested units and build a software structure dictated by design [9].

There are several methods for performing integration testing. Top-down testing starts with the topmost unit which is tested as a unit with all units it calls simulated by stubs. Stubs are simple programs designed to provide the proper response of the real unit. The proper response allows for the testing of flow of control and argument passage before further integration. Once the topmost unit is tested the stubs are replaced with the real units and they are tested with their respective stubs one at a time. This continues downward through the structure one at a time until the entire system is integrated.

Bottom-up testing is the opposite of top-down. The units at the lowest level of the software structure are tested first. The units that invoke the unit being tested are called drivers, which act in the same manner as stubs, providing proper responses. Testing moves up the structure replacing drivers with actual units until the entire system is integrated.

be very costly to fix.

2.4. Testing Steps

The first formal step is unit testing. Beizer [1] defines a unit as a system software component having the following characteristics:

(1) It is, or is intended to be, the work of one programmer.

(2) It has a documented specification that includes as a minimum: input definition, processing definition, data base definition, and interface definition.

(3) It is a visible, identifiable product that will be explicitly integrated into the program of which it is a part.

(4) It can be compiled or assembled and tested separately from other units, except such subunits as it might call.

(5) It is necessary.

The goal of unit testing is to assure that the unit performs its function according to specifications. Tests are used to uncover errors within the boundary of the unit. The units are tested separately before they are integrated.

in his eyes, successful (pride) because the software works. A test resulting in error(s) implies the engineer erred. Is this psychologically destructive? The answer lies in the objectives of software testing.

The main objective of software testing is to execute a program with the intent of uncovering an error. A successful test uncovers an error.

A second objective is to provide confidence in the correctness of the software when testing is completed. Show the software functions according to its specifications. Once testing is finished there should be reasonable confidence there are as few errors as possible.

2.3. Errors

Errors occur when the software does not perform correctly with the given input. Uncovering them is our goal. The cost of locating and fixing one can be negligible or substantial. The cost is usually in proportion to when in the software life cycle the error is discovered. One study [9] indicates that the cost of an error rises dramatically the later it is found.

The severity of an error should not be measured in terms of the cost to correct it. A catastrophic error may be easy to fix, while an unimportant functional error may errors (debug) and tests until there is reasonable confidence by some pre-established criteria that the program works.

2.2. Objectives

In school the student programmer writes software to meet the expectations of the professor. When the student tests the program he attempts to demonstrate that the software will perform its function as required. A successful test finds no errors which results in a working program and a good grade. A test that finds an error is not a successful test in the eyes of the student because he then has to spend time debugging and retesting.

The student's (and probably numerous non-student's) definition of a successful test is a contradiction of software engineering's definition. The software engineer defines a successful test as one that finds an error. Does this create a conflict with the software engineer? The engineer who develops software thinks that he has developed a great piece of software. His pride, self-esteem, and months of work are wrapped up in the software. He is ready to test to prove his work and "pass." But, he is supposed to test his program with the intent of uncovering errors, revealing faults in work. Is this a conflict of interest? The test that uncovers no errors is,

CHAPTER 2

GENERAL TESTING

2.1. Testing Introduction

This thesis focuses on software testing as a planned activity in the software engineering life cycle. It is hoped that the reader understands that proper software engineering involves testing in some form at every opportunity, whether the testing is formal or informal. A programmer who writes some code may test that code by manually executing the code at his desk. He can test the code by explaining it to others (walkthrough) or by letting others review the code in private (inspection). He can test the code by executing it on a computer with test or live data. Testing occurs when he integrates his code with other's code. The point is that testing should be a continuous process from the very beginning of software development. For our purposes, testing refers to a formal, systematic procedure.

This procedure consists of generating test cases from the program's input domain and/or invalid values, determining the expected results for each test case, executing the program on these test cases, and comparing the actual results with the expected results. Then the tester fixes

Pascal-like language structural testing approaches are adequate or whether a new approach(es) is needed for its structural testing. If a new approach is needed, guidelines for its structural testing are given in Chapter 4. The analysis includes a description of errors/problems that might occur when using the feature.

Guidelines are then given in Chapter 4 for the testing of Ada programs. These guidelines are presented through a general testing strategy and include additional structural testing guidelines for certain features. The fact that these features need additional structural testing guidelines was established in Chapter 3.

In Chapter 5, case studies are presented to demonstrate the guidelines given in Chapter 4.

Finally, Chapter 6 concludes the thesis and discusses topics for future research.

technology" has on the testing of Ada programs. This will give the "practical programmer" and tester guidance on testing his/her code and will stimulate, hopefully, the research and development of Ada testing tools and techniques. Belden [7] characterizes testing as "the most difficult, boring, and costly part of software development." Therefore, any contribution to improve testing will result in lower software costs and higher software quality.

The author assumes that the reader of this thesis has a basic knowledge of the Ada language. See the Ada Reference Manual [8] for specific Ada language information. Another assumption is made that the reader may not have an in-depth knowledge of testing approaches for Pascal-like languages or of testing in general.

A general background of testing is presented in Chapter 2 covering testing steps and current testing methods/approaches for Pascal-like languages.

Chapter 3 consists of a testing analysis of certain features of Ada. The features analyzed are selected because they potentially might require different approaches than the Pascal-like language testing approaches described in chapter two. Each feature is briefly described and then analyzed to determine whether exception handling, processing algorithms with different data types (generic program units), and concurrent real-time processing. Pascal does not support these and other features incorporated in Ada. This thesis will investigate these features from the point of view of testing.

The current literature reflects little research being done on these features or in any area of testing Ada programs. Taylor and Standish [5] are currently doing useful research on the testing of Ada tasks and in the development of the Ada programming support environment, but there are many other areas that need to be investigated. As a result, the Ada tester has no practical guidelines on the testing of Ada's features. Does the state-of-the-art of testing for Pascal-like languages apply to these features or are different testing approaches needed? Maybe some current approaches apply and maybe new approaches need to be developed. How do testing personnel test Ada programs now? Guidelines will be given. Issues and questions will be raised for future research.

According to Pyle [6], the Ada programming language represents "a major advance in programming technology, bringing together the best ideas on the subject in a coherent way designed to meet the real needs of practical programmers." The purpose of this thesis is to focus on the impac., if any, this "major advance in programming

Another way to view this is that the manner in which the specifications are implemented is being tested. This method is also called white-box testing.

The ideal goal of structural testing is to execute all logical paths in the program to demonstrate that the actual structure matches the intended structure. Unfortunately, this ideal cannot be obtained because it involves exhaustive testing. Every decision can double the number of paths and loop structures can lead to a potentially infinite number of paths. A more practical goal is to select a small but sufficient set of test paths to meet some criteria. Defining that criteria leads to several different metrics.

The simplest metric, called statement testing, is to execute all program statements. This is the most often used strategy because it is easily implemented, but is a weak measure of testing thoroughness when compared to other strategies [2]. Its main disadvantage is the possibility of executing all program statements without executing all the possible paths through the program.

DD-path testing, a stronger metric, executes every branch of decisions in a program. For example, this means to execute each direction of case statements and the THEN and ELSE of IF-THEN-ELSE statements. This is stronger than executing all statements, because all statements can be executed without executing all branches of decision statements. If all branches in a program are executed, then usually all statements will be executed. Thus, statement testing is a subset of DD-path testing.

The statements of a branch are often called DD-paths or decision-to-decision paths. Gannon [18] defines DD-paths as "the sequence of statements from the first statement of a decision branch to the next decision statement."

An even stronger metric is multiple condition testing. This is stronger because it not only means to execute the THEN and ELSE of IF-THEN-ELSE statements, but to test multiple conditions, when applicable. An example of a multiple condition is shown in Figure 2. The THEN will be executed if $x < y$ and the ELSE will be executed if $y >= x$ and $s <= r$. This satisfies the DD-path metric, but $r < s$ also needs to be tested. For this multiple condition four different test cases are required. For DD-path testing only two test cases are required. Errors due to specifications and the setting up of the multiple condition are more likely to be uncovered using the extra two test cases. For that reason multiple condition testing is a stronger metric.

```
If (x < y) or (r > s)
  Then
    ...
  Else
    ...
end if;
```

Figure 2. If Statement with Multiple Condition.

Beizer [1] proposes a fundamental path selection criteria that, as a minimum, requires enough paths to assure that every statement is executed at least once, and every decision is taken in each possible direction at least once. Note that this does not employ multiple condition testing.

Whichever strategy/criteria is used, the tester must specify the input values that cause the program to follow the desired path. Beizer [17] calls the act of finding those input values sensitizing. This is as important as selecting the path to test. This is not as easy as it might appear, because some paths may not be easily achievable. Paths not being easily achievable could be due to loops, decision variables dependent on processing, etc. The tester should select paths for testing that meet the criteria in the least complicated manner. If a path does seem to be or is unachievable, this could be an indication of an error or unnecessary code.

2.7. Testing Tools

Testing tools are used to aid in testing. One of the objectives of testing is to provide confidence in the correctness of the software when testing is completed: not a guaranteed confidence, but a reasonable confidence. Reaching that confidence point is determined by executing a set of test cases that satisfy some criteria. Tools are used to demonstrate that the software has been tested to meet that criteria.

The primary benefits that these tools are intended to provide are: precise measurement of test thoroughness; identification of program statements, DD-paths, etc., which are/are not executed during testing; description of the execution patterns associated with test scenarios; and detection of errors [19].

Tools can be broken down into the following categories: static analysis, dynamic analysis, and formal functional analysis. Static analysis refers to all processing done on source code which results in useful test and debugging information [1]. Useful debugging information is derived from the source language processor and translator. An example of useful test information which can be derived from static analysis is the identification of a program calling hierarchy. Some sort of cross reference list also results from static analysis. Static analysis does not require program execution because it operates only on the source code.

Dynamic analysis refers to processing done under execution. This is when the program is executed using a given test case to match intent with actuality. Information gathered from static analysis is used to aid in dynamic analysis.

Formal functional analysis refers to the symbolic execution of a path. This provides for the investigation of the complete functional correctness of particular paths by providing a symbolic representation of that path. One can check for the correctness of the path for all test data that will cause the path to be executed [13,20].

Each technique has its own strengths and weaknesses and are especially valuable when integrated to work together.

2.8. Automated Testing Systems

"The testing of software can be a very labor intensive, expensive, and error-prone activity. To improve the quality and cost-effectiveness of this activity, automated testing systems (ATS) have been developed. ATS encourage the use of systematic, formalized approaches to testing which ultimately improve the reliability of the software products. Automating the testing process results in lower software development costs, due to a reduction in human errors and a higher detection rate early in the development cycle, when the costs of correction are lower." [2]

The difference between ATS and the testing tools described in section 2.7 is that an ATS is a software program. An ATS is an automated testing tool.

ATS typically perform both a static analysis of the high level program and a dynamic analysis of the program's runtime behavior [2]. Static analysis functions in the same manner as already described except finding errors is left to the source language processor and translator. ATS generally require a syntactically error free program before static analysis takes place. Dynamic analysis involves the insertion of probes (data collection statements) into the program, which record execution characteristics at runtime [21].

The insertion of probes into a program to monitor runtime behavior is called instrumentation. Instrumentation is a widely used technique for monitoring execution coverage as it provides a trace of execution activities. For example, if one wants to insure that all DD-paths are executed during the testing of a program, he might insert a probe at the beginning of each DD-path to signify entry into the path (executing the first statement of a DD-path implies execution of the entire DD-path). The probes act as procedure calls which are actually embedded into the source code. The instrumented source code is then compiled and executed using given test cases. Each time a probe is reached during execution a special procedure is invoked which produces a record which later updates a data base. The data base reflects DD-path coverage. Then a report can be issued which provides DD-path coverage statistics which can show DD-paths that have/have not been executed.

This is just one example. Probes produce execution records which provide unambiguous information about the program's behavior. These records can then be processed to produce dynamic reports which reflect coverage information.

One can clearly see that the implementation of ATS incorporates the integration of static and dynamic analysis. In order to insert probes needed for dynamic analysis one must know where to insert them. This involves parsing the source code (static analysis) to determine where to put them. Then, to produce the execution records, the program must be executed (dynamic analysis) with test data.

An ATS also allows for a selective history of a program's dynamic behavior. A selective history is easier to manage, display, and hence less expensive [2]. This is done by allowing the tester to interactively specify instrumentation sites. In other words, the tester may specify what is/is not to be instrumented.

CHAPTER 3

TESTING ANALYSIS

This chapter deals with the testing implications that certain features of Ada present to the tester. This is done by analyzing certain features of Ada that potentially might require different testing approaches than those described in Chapter 2. This analysis involves several steps.

Step one was an analysis of the entire Ada language. The objective was to identify features of Ada that are different from Pascal-like languages by looking at Ada syntax charts and by researching second sources [4,22,23,24]. At this point in the selection process the testing implications of the identified features was not an issue.

Ada's Pascal-like features are not incorporated into this thesis. The reason that they are not included is that the current testing approaches for Pascal-like languages already apply in the testing of the Pascal-like features. The tester can find further information on these approaches by referring to sources found in Chapter 2 and by referring to other sources in the literature.

Step two narrowed down Ada features identified in step one by looking at each feature from a very high-level point of view of testing. This did not involve a thorough testing analysis of each feature. The criteria for selecting features for further investigation was whether there is a potential that Pascal-like language testing approaches are not adequate when testing a particular feature. If there was any doubt, the feature was selected for further study. The features not selected at this point for further study were deemed to be receptive to Pascal-like language testing approaches.

Some of the features selected at this point, like tasks and generic program units, obviously have testing implications. Pascal-like language testing approaches do not address concurrency and generic routine testing. Other features, like basic loops and overloaded operators do not at first glance seem to have any testing implications that Pascal-like language testing approaches cannot handle. Yet they were selected because they are features not incorporated in Pascal and there is a potential that current Pascal-like language testing approaches are not adequate when testing these features.

Ten features were then selected as a result of the previous two steps. The features, one at a time, are introduced and analyzed. The introduction consists of a brief description of the feature, possibly with an example of its use.

The analysis of a feature consists of a determination of whether the Pascal-like language testing approaches are adequate when testing the feature, or whether a new approach(es) is needed.

The analysis is from a structural testing point of view and not from a functional testing point of view. The functional testing perspective is not considered because there is no difference in functional testing, as described in Chapter 2, for any of the features. This is because the manner in which specifications are met is ignored in functional testing. The internal structure of the Ada program is of no concern to the tester. For example, the tester does not care if the calling of subprograms involves default parameters. The tester's concern is whether or not the actual output for a test case matches the expected output.

By analyzing each feature from a structural testing perspective, the feature is studied by looking at its actual implementation and by seeing if there are testing implications not covered in Chapter 2.

Also addressed are types of errors/problems that might occur when using a particular feature. "What can go wrong?" is investigated.

Those features that require a different structural testing approach or that have structural testing implications that the Ada tester needs to be aware of are further discussed in Chapter 4 with guidance provided for testing them.

3.1. Subprogram Default Parameters

A subprogram, which can be either a procedure or a function, consists of a specification and a body. The specification defines the subprogram's interface (calling convention) with other program units while the body contains the subprogram's implementation details. The specification defines the name of the subprogram and the parameters, if there are any.

Ada allows for a parameter defined in a subprogram specification to have a default value. This means that if a subprogram is invoked and the parameter is not given a value in the invocation, then the parameter assumes the default value given in the specification. An example of a subprogram specification with a default parameter is shown in Figure 3. The parameter ORDER has a default value of ASCENDING. The subprogram SORT can be called using the default parameter. For example, SORT can be called as:

```
SORT(MIDTERM_GRADES);
```

```
                    SORT(SOC_SEC_NUMBER);
```

In both invocations, the default value of ASCENDING will be applied to ORDER. SORT_DATA is assigned the respective passed value: either MIDTERM_GRADES or SOC_SEC_NUMBER.

```
    procedure SORT(SORT_DATA :in out NUMBERS;
                   ORDER :in DIRECTION := ASCENDING);

     ...
      If ORDER = ASCENDING
        Then
          ...
        Else
          ...
      end if;
     ...
```

Figure 3. Subprogram with Default Parameter.

Several problems/errors can occur when using subprogram default parameters. One is the potential for not executing sections of code. If every invocation of a particular subprogram in an Ada program is made using the default parameter (or not using the default parameter), then there is the possibility of not executing a section of code. For example, see Figure 3. If the subprogram body containing the IF statement is always invoked using the ORDER default value of ASCENDING, then the ELSE section of the code will never be executed. The opposite also applies. If the default parameter is never used, then the THEN section of code will never be executed. Thus, no structural testing metric can be met. However, the statement coverage metric will uncover this problem.

An error can occur that no structural testing metric will uncover. This occurs when the subprogram default value introduces an error such as divide by zero. For example, see Figure 4. If the default value 5 of DIVISOR is used, a divide by zero error will occur. If the default value is never used in the execution of the CALC procedure, the divide by zero error caused by the default value will never be uncovered. This error will not be caught by any structural testing metric because the path is executed. This meets the statement coverage/DD-path testing metric. It will be uncovered only if the default value is used.

```
procedure CALC(VAL1 :in out NUM;
               VAL2 :in out NUM;
               DIVISOR :in NUM := 5);
 ...
 For INDEX in 1..DIVISOR
   loop
     ...
     VAL1 := VAL2 / (5 - INDEX);
     ...
   end loop;
 ...
```

Figure 4. Divide by Zero Error.

3.9. Exception Handling

In many applications, such as weapon systems or transportation systems, it is critical that the system be able to recover from erroneous conditions. Ada provides this capability through its exception handling facility. An exception is defined as "a condition necessitating suspension of normal program execution" [23]. An exception handler is a section of code to be executed when an exception occurs.

There are two types of exceptions in Ada: predefined and user-defined. Predefined exceptions are declared in the package STANDARD. They are raised (bring attention to the condition) implicitly by the run-time system, although they can be raised explicitly via the RAISE statement. It is difficult to predict where and when they will be raised because they are implicitly raised.

User-defined exceptions are created by the programmer. They must be raised explicitly by a raise statement. It can be determined when and where they will be raised because they are explicitly raised.

When an exception is raised, whether implicitly or explicitly, it is handled by an appropriate exception handler. If an exception handler is not present in the body of the current subprogram, package, or task, then see Figure 8. ROWS and COLUMNS are given the default values of 66 and 132. The problems/errors that can occur when using generic parameter default values are the same problems/errors described for subprogram default parameters in section 3.1. The tester needs to look at the Ada program to see if they are being used.

```
generic
  ROWS    : in INTEGER :=  66;
  COLUMNS : in INTEGER := 132;
package OUT_DATA is
  ....
```

Figure 8. Generic Parameter Default Values.

Ada also permits the passing of subprograms as parameters to generic program units. The programmer can declare a default subprogram for a generic subprogram parameter. Again, see section 3.1 for potential problems/errors when using default parameters and, as for generic parameter default values, the tester needs to know if default subprograms are being used.

Guidelines for the structural testing of generic program units are given in Chapter 4.

The programmer decides to use the generic sort routine to meet the requirement. He instantiates the generic sort with an array size of one thousand. The tester then reasons that testing of the call number instantiation is not necessary because the sort generic routine has already been tested.

The program goes into production. One day only one new library book is received and the sort routine fails, bringing the system crashing down. The generic routine was never tested for a single element sort. That test was not possible for the sort by day of the week instantiation because there was always seven days sorted. The test would have been possible for the call number instantiation. The routine was tested for the maximum array size but not for the minimum array size.

Generic routines involving mathematical calculations are very suspectable to error if each instantiation of the generic routine is not tested. Exponentiation and division (i.e. divide by zero) statements can easily result in numeric errors.

The generic program unit can create other problems. Ada permits the definition of default values as generic formal parameters. The definition of such a parameter takes the form of a variable declaration. For example,

ple for routines of larger size.

A second problem is that no structural testing metric can be met if only one of several instantiations of the same generic program unit is tested. Entire sections of code are not tested.

A significant problem is that there is no absolute guarantee that the generic routine will function without error for each instantiation. This is despite the fact that the programmer can define generic parameter types, thereby limiting the use of a generic routine to predetermined types. For example, a generic sort routine has been written with the primary objective of sorting by day of the week. The sort key has been designated "type ELEMENT is private", which means that the routine will sort any set of data that permits assignment and test for (in)equality. The generic array that holds the data to be sorted is an unconstrained array. This means that its size can vary with each instantiation.

An instance of the generic routine is created to sort by day of the week with an array size of seven. Assume there is always seven days of data to sort. The instantiation is tested and deemed ready for use.

Management then decides that it wants to be able to sort new library books received each day by call number.

This definition of a generic program unit is just a template. It does not create any code. Therefore, it is not executable and cannot be used directly. An instance of the generic program unit must be created in order to use it.

The act of creating an instance of a generic program unit is called instantiation. When an instantiation occurs, the generic template is "filled in" with information from the template and the program continues as if a regular subprogram or package has been defined.

Generic program units provide several error/problem areas for the Ada tester. One is the question of whether or not each instantiation needs to be tested. The argument can be raised that if the generic program unit is truly generic, then testing is needed for only one instantiation. The testing of other instantiations of the same generic program unit is not needed because the generic algorithm is logically the same for all instantiations. If it works for one instantiation, it will work for all instantiations!

There are several problems with this reasoning. One problem is establishing that a generic routine is truly generic. This may be simple for an exchange routine which consists of only three statements, but it is not that sim-

The use of unchecked programming has no structural testing implications. They are not testable. To use either generic subprogram the programmer simply states that one/both are in effect via the WITH statement. No testable code is generated. The tester should be sure to test code that unchecked programming applies to with the strongest feasible testing criteria.

Unchecked programming is not addressed in Chapter 4.

3.8. Generic Program Units

Generic program units are algorithms that can be used with different data types. They allow the Ada programmer the ability to factor out common algorithms used on different data types. This helps manage the complexity of programs. For example, a programmer may want to exchange two elements of the following types: integer, enumerated, and real. In Pascal-like languages three subroutines are required, one for each type. In Ada one generic subprogram is required.

To create a generic program unit a prefix, called the generic part, is added to a subprogram or package specification. This defines the specification as being generic and defines all generic parameters, if any. Then the generic body is added.

Low-level statements provide no additional structural testing implications. This feature is not included in Chapter 4.

3.7. Unchecked Programming

Unchecked programming allows the relaxation of Ada's typing and elaboration rules. It is done through the execution of two predefined generic subprograms: UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION.

The UNCHECKED_DEALLOCATION generic subprogram is used to explicitly deallocate the space for a dynamic element. There is no automatic garbage collection by the system. The danger in doing this is in the deallocation of an element that has other elements pointing to it. Ada makes no guarantees concerning the value of the pointer of the element(s) pointing to the deallocated element. The element now points to a nonexistent (deallocated) element.

The UNCHECKED_CONVERSION generic subprogram permits unrestricted conversion from one type to another. This is necessary if there needs to be an operation between two incompatible types. The danger in doing this is that there is no guarantee concerning the result. The programmer must ensure that the properties of the target type are maintained and that the results are correct.

operator for a type for which the operator is not predefined, he/she must write an appropriate overloaded subprogram using the expressive operator as the name of the subprogram. For example, if a programmer wants to use the operator "+" to perform a set operation, he/she can redefine it as the name of a function which performs the desired set operation. This is called overloading an operator. The compiler determines the proper subprogram invocation by examining the invocation parameters.

The only problem that overloading operators presents to the tester is possible confusion during debugging and that has nothing to do with the actual testing of the program. They have no effect on the structural testing of Ada programs and are not a part of Chapter 4.

3.6. Low-Level Statements

Ada allows for the writing of low-level (assembly) code. This allows the Ada programmer to avoid going outside of Ada when low-level programming is necessary.

The low-level statements are placed in a subprogram that contains only low-level statements or declarations. A package must be defined that exports a record abstracting the low-level instruction set.

world. The only operations that may be performed on STACK_RECORD are PUSH and POP. The user may not CLEAR the stack or manipulate it through STACK_RECORD in any way. The point is that the implementation details of procedures PUSH and POP would not be any different if they were not in the package. The code is logically the same. Thus, the structural testing of PUSH and POP, which would be contained in the body of the STACK package, does not require a different structural testing approach.

```
package STACK is
  type STACK_RECORD is private;
  procedure PUSH( parameter list );
  procedure POP ( parameter list );
private
  .....
  type STACK_RECORD is
    record
      .....
    end record;
end stack;
```

Figure 7. Package Specification.

Packages need no special structural testing guidelines and are not discussed in Chapter 4.

3.5. Overloading Operators

Expressive operators are predefined in Ada for only certain types. This is because Ada is a strongly typed language. If the programmer wants to use an expressive ponents." Booch [4] defines a package as "a collection of computational resources, which may encapsulate data types, data objects, subprograms, tasks, or even other packages." A package encapsulates or puts a wall around these resources.

The main objective of a package is to support the software engineering principles of information hiding and data abstraction. Packages do this by giving the programmer a way to physically group related entities into a logical chunk. This is done through a package's specification and body. The specification lists the visible (to other program units) parts of the package. The implementation details of the package are in the body. These details are not visible outside of the package. It is not important for the user to know how the operations are actually implemented.

Does this "most important improvement over Pascal" have structural testing implications? The answer is no because a package is simply a collection of resources. Its purpose is not to change how resources are implemented, but to hide how they are actually implemented. For example, in Figure 7 a package called STACK is defined having two possible operations: the procedures PUSH and POP. The STACK_RECORD of the stack is a private type meaning that its structure is not visible to the outside

The pragma SUPPRESS is used in Ada to suppress various run-time checks which detect exceptional conditions. The act of detecting exceptional conditions has a slight runtime overhead, so if there is some reason to do so (efficiency), these checks may be suppressed. One application in which the use of this pragma is tempting is the real-time application. However, one of the main reasons that the exception handling feature is a part of Ada is so real-time systems do not fail during execution. They can recover through an exception handler. If the pragma SUPPRESS is in effect, recovery via an exception handler is not possible.

This feature has no testing implications as far as the actual testing of the program is concerned. However, some points need to be made. Make sure that the section of code that the pragma SUPPRESS applies to has been thoroughly tested, especially if it is critical that the program does not fail. Do not use this feature unless there is a justifiable reason to do so.

3.4. Packages

A package is one of Ada's fundamental program units. According to Feuer and Gehani [22] "packages are probably the most important improvement of Ada over Pascal because they allow for the effective exploitation of program comto the end of the loop is never executed. No structural criteria can be met.

```
loop
   ...
   exit when WEIGHT > MAX or READY_TO_LOAD;
   ...
end loop;
```

Figure 6. Conditional Exit Statement in Basic Loop.

There is a potential for boundary errors or errors due to the use of multiple conditions. Both of these problems are centered at the conditional EXIT statement.

The above problems/errors do not require different structural testing approaches. The structural testing metric of statement coverage will uncover the non-execution of code. Multiple condition testing will uncover multiple condition/boundary errors.

The basic loop has no effect on the structural testing of Ada programs and is not included in Chapter 4.

3.3. Pragma SUPPRESS

A pragma is an instruction to the compiler. One pragma in particular, the pragma SUPPRESS, is investigated here.

3.2. The Basic Loop

Ada has three control structures that allow for a section of code to be executed zero or more times. One of the structures is called the basic loop (the for loop and while loop are the others). It is structured to loop forever. To leave the loop, the EXIT statement must be used. The EXIT statement can be conditional or unconditional.

The basic loop has two common uses in Ada. It is used as the outer loop of a task that must continue forever once it is initiated. It is also used in conjunction with an exception handler when interactive data is being input. The section of code contained in the basic loop repeats until proper data is input.

There are problems/errors that can occur as a result of using basic loops. The non-execution of a section of code within the basic loop is one problem. This can occur when an EXIT statement is not the last line of the basic loop. If the EXIT statement is unconditional then the rest of the code in the basic loop is definitely not executable. If the EXIT statement is conditional (uses WHEN clause) then there is the possibility of code not being executed. For example, see Figure 6. If the WEIGHT is greater than MAX or READY_TO_LOAD is true every time the loop is entered, then the code from the conditional EXIT

Another problem area is the use of multiple subprogram default parameters. This could easily lead to multiple condition statements where the potential for error is great. For example, see Figure 5. DV1 and DV2 can have default values when the subprogram containing the IF statement is called. The multiple condition structural testing metric can be used to test this section of code by using four different test cases to cause each of the four possible conditions of the IF statement to occur. The problem is that a test case does not have to include the default values of both DV1 and DV2. If both default values are used at the same time in this example the execution of the exponentiation statement could result in a NUMERIC_ERROR.

```
...
If (DV1 > X) and (DV2 > Y)
  Then
    ...
    X := DV1 ** DV2;
    ...
  Else
    ...
```

Figure 5. Multiple Subprogram Default Values.

It can be seen that the use of subroutine default parameters do have structural testing implications. Thus, guidelines for testing them are given in Chapter 4.

control passes up (propagates) to the next level until either an exception handler is found or the operating system is reached. The exception handler acts according to the desires of the programmer. There are no predefined exception handlers. It is the responsibility of the programmer to respond to an exception with an appropriate algorithm.

Several errors/problem can occur when using the exception handling facility. One problem involves the use of predefined exceptions. The potential of their occurrence adds considerable complexity to structural testing if the desired metric is DD-path testing. The problem is that they create additional implicit DD-paths. To find all such paths requires that for each statement in the program, one determines all exceptions, if any, that might be raised by the execution of that statement. It is not feasible for the tester to do that.

Another problem areas is the propagation of exceptions. Tripathi et al [24] notes that "a relatively high level routine may potentially be the recipient of a multitude of exceptions propagating from lower level environments." This creates the same problem stated in preceeding paragraph: that of additional DD-paths.

It also complicates the integration testing of high level routines. Should stubs include possible exceptions propagating from below?

Another problem caused by propagating exceptions is that the principles of data abstraction and information hiding may be violated. This is inconsistent with the static scoping of the rest of the language. Many propagated exceptions, specific to lower level routines, have little relevance at the higher level. Privacy is compromised. For example, propagating from a package might give details of the package's implementation which otherwise are hidden.

A big problem/question concerns exceptions and parameter passing. For a subprogram with out or in-out parameters, the effect of propagating exceptions on its environment may be undefined [24]. For example, see Figure 9. The statement "Y := Y * Y" will in all probability raise NUMERIC_ERROR which will be propagated upward because there is no NUMERIC_ERROR exception handler in CALCULATE. The value of the actual parameter corresponding to X may be unchanged or may have increased by some amount, depending on the particular parameter passing mechanism. How does the programmer write the proper exception handler or the tester know if proper recovery has taken place if the values of parameters are unknown?

```
procedure CALCULATE(X : in out INTEGER;
                    Y : in INTEGER) is
  COUNT : INTEGER;
  begin
    for COUNT in 1..500
      loop
        X := X + 1;
        Y := Y * Y;
      end loop;
  end CALCULATE;
```

Figure 9. Propagating Exceptions.

By this time, one may wonder whether exception handlers should even be tested. The answer is yes because Ada programs do not have to include them. If they are present in a program, they are there for a reason. The programmer wants a certain response (exception handler) to a certain situation. The tester must test the exception handler to make sure the response is correct: that the program recovers correctly. Remember that predefined exceptions do not give predefined answers. If it is not critical to recover, then maybe exception handlers should not be in the program in the first place.

Exception handling confronts the Ada tester with many testing problems. Pascal-like language approaches do not consider these problems. For that reason, guidelines are given in Chapter 4 for the structural testing of exception handlers.

3.10. Tasks

The task is the Ada feature that supports concurrent real-time processing. It allows for a number of different activities to progress in parallel. A task consists of two parts, a task specification and the task body. The specification defines the interface between the task and other units. The body defines the action of the task.

Tasks are based on the concept of communicating sequential processes. They can be viewed as independent, concurrent activities that communicate with each other via messages. Two tasks rendevous when one passes a message to the other. The task receiving the message accepts the message from the calling task. The calling task is said to enter the receiving task. The statements that allow for this communication are the ENTRY and ACCEPT statements.

If one task gets to the entry point before the other, the first task will put itself to sleep (suspend itself) until the other task arrives at the rendevous point. If the programmer wants the waiting task to do something else, several options are available.

Tasks present many problems/error possibilities that need to be tested for. One error is the synchronization error. The synchronization error results from sequencing problems. Examples of such failures given by Leveson and Stolzy [25] include 1) incorrect sequences such as not having statement X occur before statement Y or not having rendevous A occur before rendevous B, and 2) not reaching a specific statement such as not reaching a task termination statement.

Another unwanted situation that can arise from sequencing problems is deadlock. A deadlock occurs when resources are tied up by activities to the point where further processing is blocked. For example, process A wants resources that process B has and process B has resources that process A has in its possession. Both processes are waiting for resources that can never become free. Neither can proceed resulting in deadlock. Therefore, the program cannot proceed. This situation must be uncovered during testing.

Another problem for the tester is executing all possible synchronizations that may occur during execution. In other words, coming up with all possible sequences of concurrency related events. Is this feasible? In tasks involving dynamic entities (pointers, subscripts) the number of synchronization scenarios can get to be extremely large. Some intra-task paths may not even be executable which may be very difficult to determine.

The DELAY statement suspends task processing for an exact amount of time. Should its presence be considered when deriving the scheduler algorithm? If so, the tester must know that it is present in the program. Ada also allows a task to quit an attempted rendevous after some maximum delay through a construct known as a timed entry call. The problem in doing this is the danger of never reaching the maximum delay time and never executing statements whose execution depends on reaching the maximum delay.

All of the above problems necessitates looking at the actual Ada code for testing. Pascal-like language testing approaches do not address the above issues. Testing guidelines are given in Chapter 4.

3.11. Summary

Four of the ten features are given structural testing guidelines in Chapter 4. The features are subprogram default parameters, generic program units, exception handling, and tasks. These features require different structural testing approaches or have structural testing implications not covered in Chapter 2.

The other six features are not addressed in Chapter 4. Pascal-like language structural testing approaches are adequate for the structural testing of these features. hx

CHAPTER 4

TESTING GUIDELINES

In this chapter testing guidelines are given for Ada programs. These guidelines are presented through a general testing strategy and include additional structural testing guidelines for certain Ada features.

The general testing strategy applies to Ada as a whole. Additional structural testing guidelines for certain Ada features are presented. They are to be used as a supplement to the general testing strategy. These guidelines are needed because Pascal-like language structural testing approaches are not adequate for the structural testing of these features. This was established in Chapter 3. The guidelines address the problems/questions raised in Chapter 3 for each feature.

The features receiving the additional guidelines are subprogram default parameters, generic program units, exception handling, and tasks.

4.1. General Testing Strategy

The general testing strategy is based on the testing methods and testing steps of Chapter 2. This entails using the testing methods of functional testing and structural testing. For additional information on test data design see [26]. Structural testing is based on a Pascal-like language structural testing approach. It should be supplemented with the structural guidelines given below for certain Ada features.

Some of the supplemental guidelines are based on the assumption that a structural testing statement coverage metric may not be met (i.e. statement coverage). These guidelines could be already satisfied if a metric is met. They will be annotated to reflect that fact.

Some supplemental guidelines can be met by a Pascal-like language testing metric. They are included because not everyone will test to meet a structural testing metric. It is important that a structural testing metric be met to properly test the feature. For that reason, those guidelines are given. If a structural testing metric is met during testing, then guidelines that can be satisfied by meeting that structural testing metric can be ignored.

The general testing strategy involves four testing steps. They are unit testing, integration testing, validation testing, and system/acceptance testing.

Perform unit testing using the testing methods of Chapter 2. If the unit test involves subprogram default parameters, generic program units, exception handling, or tasks, then supplement the unit testing with the appropriate additional structural testing guidelines. The developer of the code should perform unit testing.

The next testing step is integration testing. Perform builds and threads integration testing with a bottom-up emphasis. This means integrating threads of routines starting at the bottom level of the program and working up from there. Use the functional testing method of Chapter 2. Structural testing in this step is used to insure that each interface (invocation) is executed. Each unit of code does not have to meet a structural testing metric. That should have been accomplished during unit testing. If the code being integrated involves generic program units, exception handling, or tasks, then supplement the structural testing with the appropriate additional structural testing guidelines. A test team should perform integration testing.

The final steps are validation and system/acceptance testing. Both are performed using only functional testing test cases. No supplemental testing guidelines are necessary. A test team from the selling organization should perform validation testing. A test team from the buying organization should perform system/acceptance testing.

4.2. Subprogram Default Parameters

The testing guidelines given for subprogram default parameters are for the subprogram developer at the unit testing level. Testing is done at the unit testing level because the concern is to uncover errors within the boundary of the subprogram. Unit testing is done through the use of simulated invocations.

Simulated invocations are accomplished by drivers which provide input to the unit being tested. They allow the tester the capability to test a particular subprogram default value or a combination of subprogram default values for the given subprogram. The tester can derive as many test cases as necessary without knowledge or caring whether or not a subprogram default value or a combination of default values is always/never used in actual invocations of the subprogram. This is not possible during validation, system/acceptance testing or integration testing not involving drivers because invocations cannot, without recompilation, be altered. If a subprogram default value is not used in any invocation of a subprogram during system testing there is nothing the tester can do to test it unless he changes the code.

The tester should look at the subprogram specification of the subprogram to be tested and determine if a subprogram default value(s) is declared. If a default parameter(s) is declared then that value(s) should be used in a test case(s) as described below.

Execute all the subprogram default values together in one test case. This is justifiable because an invocation of the subprogram can default to all of the default values. This test case will uncover errors that occur only if all the default values are used together. For example, see Figure 10. Procedure CALC performs scientific calculations that involve variables that can have default values when CALC is invoked. If all the default values are used at the same time, a divide by zero error will occur when computing RESULT1. This error is uncovered only if all of the default values are used together.

```
procedure CALC (DV1: in NUM := 1;
                DV2: in NUM := 2;
                DV3: in NUM := 20;
                DV4: in NUM := 40;
                RESULT1 : out NUM;
                RESULT2 : out NUM;
                RESULT3 : out NUM) is
 ....
 RESULT1 := DV4 / ((DV1 * DV2 * DV3) - DV4);
 ....
 if (DV1 /= 1) and (DV2 /= 2) and (DV4 /= 40)
   then
     for INDEX in 1..DV3
       loop
       RESULT2 := RESULT2 / (20 - INDEX);
       end loop;
 end if;
 ....
 if (DV1 /= 1) and (DV2 /= 2)
  then
    RESULT3 := DV3 ** DV4;
 ....
end CALC;
```

Figure 10. Errors Caused by Default Values.

Next test each default value one at a time. If there are four default values, as in Figure 10, then this translates into four test cases. This uncovers an error that occurs only if a test case is executed comprised of the one default value. An example of this is shown in Figure 10. The statement where RESULT2 is calculated will result in a divide by zero error only if DV3 defaults to its default value and DV1, DV2, and DV4 do not default to their default values.

The four default parameters in Figure 10 translate into 2 to the 4th power minus 1 possible test cases if all combinations are tested. That computes to fifteen test cases. The subtracted one represents the test case with no default values. Of the fifteen possible test cases, five are already accounted for. One is the test case of all default values and the other four are the test cases of one default value each. This leaves ten additional test cases that can be executed. These test cases are comprised of other combinations of the default values.

It can be shown that each remaining untested combination of default values can result in an error that would otherwise remain uncovered. For example, in Figure 10 if a test case uses the default values of DV3 and DV4, and does not use the default values of DV1 and DV2, the calculation of RESULT3 would result in a NUMERIC_ERROR. If this test case had not been executed, this error would remain hidden.

In the case of the four default values it is certainly feasible and practical to execute the additional ten test cases. However, if a subprogram has eight default values the total possible test cases is 255 (2 to the 8th power minus 1). This number is reduced to 246 by executing the test case of all default values and by executing the eight test cases of one default value each. It is not practical to execute the remaining 246 test cases.

It is recommended that if there are four or less subprogram default values, then execute the remaining combinations of default values. This translates into a maximum of ten additional test cases. If there are five or more default values, stop testing after the first two steps. Five default values translate into 31 possible test cases. Subtracting one for the test case of all default values and five for the test cases of one default value each leaves the possibility of executing 25 additional test cases. Draw the line here on executing the additional test cases unless other information suggests otherwise.

There may be errors that are only uncovered by executing the additional test cases, but realize that there is more testing to be done. Integration testing will test the interfaces to the subprogram. At that time all invocations of the subprogram will be executed. Validation and system/acceptance testing will also occur. The point is that errors caused only by the use of certain combinations of default values can be uncovered in other phases of testing.

Following is a summary of the guidelines for the structural testing of subprogram default values.

Execute each RAISE statement that causes an exception to be propagated to the exception handler being tested. This takes care of isolated RAISE statements within lower level exception handlers and RAISE statements for user-defined exceptions.

If the exception handler has been designed to handle a predefined propagating exception, then strive to raise one predefined exception from each lower level routine. Do not attempt to raise each possible occurrence of the exception in each routine. That is not practical. Raising one exception from each routine is sufficient. Do not be concerned with meeting a structural testing metric in the exception handling routine. That testing should have occurred during unit testing.

Meeting this criteria may not always be practical. It may not be practical when lower level routines are several levels down in the calling hierarchy. Deriving a test case that will cause an exception to be raised from that lower level routine may be very difficult and time-consuming. Use good judgement. If it is not practical to raise an exception from a lower level routine, then do not do it.

An example of these guidelines is illustrated in Figure 13. Both the HIGHER_LEVEL procedure and the NESTED thread is integrated, the test team should determine if the exception handlers in the thread handle propagating exceptions. Each exception handler that: 1) has in place all lower level routines in the calling hierarchy, and 2) has not been tested yet, should then be tested as described below.

All lower level routines that are in the calling hierarchy of the routine containing the exception handler should already be in place. This is necessary because no lower level routine should be represented as a stub. This includes routines not involved in propagating the exception that might have values affected by the recovery. The status of all variables involved is under test. Using stubs will not provide answers on the effects on variables within the code the stub is simulating. Values passed as parameters can have far-reaching effects on other variables throughout the program.

There is one important rule for the structural testing of propagating exceptions. Test only exceptions that a higher level exception handler has been designed to handle. If the tester starts testing exceptions at random just "to see what happens", then there is no criteria for stopping. Do not start. Stick to testing exceptions that have been planned for.

satisfy the DD-path metric, then raise as many predefined exceptions as necessary to meet that metric.

```
.... {start of a block}
declare
  OVERFLOW_LEVEL : exception;
begin
  ....
  if FLUID > MAX_LEVEL
    then
      raise OVERFLOW_LEVEL;
  ....
  if FLUID2 > MAX_LEVEL
    then
      raise OVERFLOW_LEVEL;
  ....
  exception
    when OVERFLOW_LEVEL =>
      OPEN_VALVE;
      ALARM;
    when NUMERIC_ERROR =>
      FIX_IT;
    when CONSTRAINT_ERROR =>
      RAISE;
    when others =>
      LOG_ERROR;
  end;
end;
```

Figure 12. Unit Testing of Exception Handler.

The structural testing of exception handlers for propagating exceptions is performed by a test team and should be done at the integration testing level because more than one level of code is involved. Bottom-up integration testing is ideal, but the recommended builds and threads technique certainly facilitates this testing effort. Integrate threads in a bottom-up fashion. Each time a new structural testing metric.

An example of these guidelines is illustrated in Figure 12. Recall that testing is done at the unit testing level. Do not consider propagating exceptions from a lower level routine. In this example there are four WHEN clauses in the exception handler. A WHEN clause designates the action in response to a particular exception. Statement coverage is an appropriate metric in this case because each exception handling routine is sequential in nature. If exception handling routines contain DD-paths, then consider using the DD-path metric as the structural testing criteria.

Execute each WHEN clause and each RAISE statement within the body of the routine declaring the exception handler at least one time. In Figure 12 one WHEN clause contains the recovery routine for the predefined exception OVERFLOW_ERROR. There are two explicit RAISE statements that raise it. Execute both RAISE statements. The other three WHEN clauses have to be raised implicitly. Do not attempt to raise all possible exceptions that cause their execution. Raise a predefined exception for each WHEN clause one at a time. This is all that is necessary in this example in order to meet the statement coverage metric. If there would have been multiple DD-paths in the example exception handler routines and the tester wants to

Do consider RAISE statements that cause an exception to be propagated upward from the unit of code being tested. These RAISE statements cause exceptions that cannot be handled by the exception handler undergoing unit testing. Execute each of these RAISE statements. It is necessary to do this because the tester wants to make sure that each RAISE statement can be reached. Do not attempt to implement (via a driver) an exception handler to accept the propagating raise. Let the code fail. Propagating raises will be tested during integration testing.

Make sure that each RAISE statement, which invokes the exception handler undergoing unit testing, is executed even if several raise the same exception. It is important to test if the program recovers from each erroneous condition for which there is a RAISE statement.

If a predefined exception has been declared, do not attempt to raise each possible exception that causes its exception handler to be executed. That attempt should not be made because predefined exceptions are implicitly raised. This means that they can be raised from many different statements in many different ways. The bottom line is that predefined exceptions create implicit control paths. To try to determine each control path is not practical. Raise as many predefined exceptions as necessary to have their respective exception handlers meet a appropriate test instantiation(s), but do not test these as extensively as the first test instantiation. Use the same invocation that was created to test the initial test instantiation to test these instantiation(s). It is recommended that the tester perform functional testing, but do not perform boundary testing or meet a structural testing metric.

Following is a summary of the guidelines for the structural testing of generic program units at the integration level.

(1) Determine if an instantiation has taken place for a generic program unit in any of the thread routines integrated. If so, test it using functional testing test cases.

4.4. Exception Handling

These structural testing guidelines for exception handlers are given for unit testing and integration testing.

The first focus is on unit testing guidelines. The structural testing in this case should be done at the unit testing level by the developer of the exception handler. Do not consider the possibility of propagating exceptions from a lower level routine.

(1) Identify the types of the generic parameters. These types are found in the generic part. Check Figure 11 to determine compatible actual parameters for each generic type. Create a test instantiation with no default values using Figure 11 as a reference to determine data types. The data types declared in the instantiation should be of type integer whenever possible.

(2) Create an invocation to test the test instantiation.

(3) Perform functional testing. Be sure and perform thorough boundary testing as this is the only instantiation in which boundary testing will occur. This is part of the general testing strategy but cannot occur until the first guidelines have been followed. Next test the generic routine using a structural testing metric. This will be satisfied by meeting the statement coverage metric.

(4) Determine if generic default parameters or generic subprogram default parameters are defined in the generic part. If so, test those default values following the guidelines given for subprogram default parameters. Note that a test case for subprogram default parameters translates into a test instantiation for generic default values. Create the

never change once instantiated. Do not test boundary values or the structure of the generic routine again. That has already occurred.

Generic program units should also be tested during integration testing. Thorough structural testing, logic testing, and boundary testing occurred during unit testing so it is not necessary to do those types of testing again.

The integration testing of a generic routine is analogous to the integration testing of a unit tested procedure. The concern at this point is on interface errors and functional validity.

It is recommended that the generic program unit, when instantiated during integration testing, be integrated using functional testing test cases. This guideline recommends using functional testing when these are supposed to be structural guidelines. The reason this guideline is given is so the test team will know what to do when they come across an instantiation during integration testing.

Following is a summary of the guidelines for the structural testing of generic program units at the unit testing level.

instantiation of a generic routine. However, the error was not because of the data type given for the actual parameter. The error was a result of not testing the sorting of one element. This was a logic error that should have been uncovered during unit testing by logic testing. If the generic routine would have been tested during unit testing for the sorting of one element, regardless of the type of data being sorted, the error would have been uncovered. Instead it was uncovered after the program was operational.

If no default values are present, unit testing is complete. If default values are present, then additional test instantiation(s) are recommended. Test the generic default values/generic subprogram default values by following the guidelines given for the unit testing of subprogram default parameters in the previous section. For example, the first instantiation involving defaults will consist of all the default values. This parallels guideline #1 for subprogram default parameters.

For each test instantiation at this point, use the same functional testing strategy used during the testing of the initial test instantiation. The same test cases could possibly be used again. The only test data that will change are the default values and they are not truly test data because they are constants. Those values can large size. Use the minimum and maximum integer values for array bounds if the array is unconstrained. The point is to do boundary testing and integer values facilitate this type of testing.

Integer types cannot be used for generic types #3, #6, and #7 shown in Figure 11. The tester must use the actual type indicated. For example, the floating point data type must be used for generic type #7. Again, derive test cases to test boundaries.

The above guidelines are given concerning recommended data types to use during testing. They provide guidance on the question of how many instantiations should be tested during unit testing. The developer cannot possibly anticipate how the generic routine will be instantiated in every case. If the generic type parameter #1 is supposed to work for any data type, then test one data type for it in the initial test instantiation. From Figure 11 it is recommended that the integer data type be used for generic type #1. The integer type will then be used in the initial test instantiation.

The example given in section 3.8 of Chapter 3 concerning there being no guarantee that a generic routine will function without error for every instantiation is true. In the example, an error is uncovered after a new

```
(1) type GENERAL_PURPOSE is limited private;
    - matches any data type
    - use integer type
(2) type ELEMENT is private;
    - matches any type that permits assignment and
      test for inequality
    - use integer type
(3) type LINK is access OBJECT;
    - matches any access type designating the same
      type of object
    - use access type
(4) type ENUMERATION is (<>);
    - matches any discrete type
    - use integer type
(5) type INTEGER_ELEMENT is range <>;
    - matches any integer type
    - use integer type
(6) type FIXED_ELEMENT is delta <>;
    - matches any fixed point type
    - use fixed point type
(7) type FLOAT_ELEMENT is digits <>;
    - matches any floating point type
    - use floating point type
(8) type CONSTRAINED is array (INDEX) of ELEMENT;
    - matches any constrained array of the same
      dimensions, index types, and type of components
    - use integer type for INDEX
      use either predetermined type or integer type
      for ELEMENT
(9) type UNCONSTRAINED is array(X range<>) of ELEMENT;
    - matches any unconstrained array of the same
      dimensions, index types, and types of components
    - use integer type for X
      use either predetermined type or integer type
      for ELEMENT
```

Figure 11. Generic Type Parameters.

It is recommended that the tester use integer types whenever possible in the test instantiation. Then use boundary values like the maximum integer or the minimum integer in test cases. If a generic type array is declared then test an array of size one and an array of a

The data types used in this test instantiation depend on the types of the generic parameters. For example, see Figure 11 [4]. It contains a list of all generic type parameters. With each type a compatible actual parameter(s) is defined. For example, generic type parameter #1 in the Figure matches any data type. This means that any type of data is allowed for this generic type during an instantiation of the generic routine.

whether or not default subprograms are declared for for generic subprogram parameters. This information is used to derive test instantiations and to determine how many test instantiation to derive.

Test instantiation(s) are used to test the generic routine. The number of test instantiations needed to test the generic routine depends on the number of default generic parameters/generic subprogram parameters. If no defaults are present then only one test instantiation is necessary. If defaults are present, then more than one test instantiation is necessary.

The first test instantiation involves no default values. If no default values are defined, then this is not an issue. This instantiation will serve as the only test instantiation needed.

The tester should execute test cases that enable a structural metric to be met. Test cases should also thoroughly test the logic of the routine and incorporate boundary testing. A case study is provided in Chapter 5 that demonstrates these test cases. This is the only instantiation in which the logic is to be thoroughly tested. Test data is input via a driver that simulates an invocation of the generic routine.

(1) Execute one test case consisting of all default values.

(2) Execute test cases consisting of one default value at a time.

(3) If there are four subprogram default values or less, then execute test cases consisting of all remaining combinations of default values. This is a maximum of ten additional test cases.

(4) If there are more than four default values, do not execute any more test cases. To execute all remaining combinations of default values translates into a minimum of 25 additional test cases.

4.3. Generic Program Units

The testing guidelines for generic program units are given for unit testing and integration testing. The developer of the generic routine performs unit testing and a test team performs integration testing.

When the generic routine is unit tested, the tester should investigate the "generic part" of the routine. Recall that it is a prefix to a subprogram or package specification. Three pieces of information should be derived: the type of each generic parameter, whether or not default values are defined for generic parameters, and procedure have already been unit tested. The procedure being tested in this example is the HIGHER_LEVEL procedure because it has been designed to handle exceptions propagating from the NESTED procedure. Assume that the HIGHER_LEVEL exception handler has been designed to handle only propagating exceptions of types OVERFLOW_LEVEL and CONSTRAINT_ERROR.

```
procedure HIGHER_LEVEL is
  OVERFLOW_LEVEL : exception;
  procedure NESTED(X : in NUM);
  ....
  procedure NESTED(X : in NUM) is
    LOCAL_ERROR : exception;
    ....
  begin {NESTED}
    ....
    if FLUID > MAX_LEVEL
      then
        raise OVERFLOW_LEVEL;
    ....
    exception
      when LOCAL_ERROR =>
        FIX_LOCAL;
      when CONSTRAINT_ERROR =>
        raise;
    end;
  end NESTED;

begin {HIGHER_LEVEL}
  ....
  exception
    when CONSTRAINT_ERROR =>
      FIX_GLOBAL;
    when OVERFLOW_LEVEL =>
      OPEN_VALVE;
      ALARM;
    when NUMERIC_ERROR =>
      FIX_NUMERIC;
  end;
end HIGHER_LEVEL;
```

Figure 13. Integration Test of Exception Handler.

The test team should execute the explicit RAISE OVERFLOW_FLOW statement in the NESTED procedure. If there were more, then each would be executed. The test team should also implicitly raise one CONSTRAINT_ERROR exception in the NESTED procedure. If there were a nested procedure within the example NESTED procedure, then one CONSTRAINT_ERROR exception would be raised from that procedure also. Both the CONSTRAINT_ERROR and OVERFLOW_ERROR exceptions are raised because they are designed to be handled in the HIGHER_LEVEL exception handler. It is not necessary to raise a predefined NUMERIC_ERROR exception from the NESTED procedure because the HIGHER_LEVEL exception handler is not designed to handle a propagating exception of that type.

The strategy for testing exception handlers differs with respect to the testing step. The developer should test it at the unit testing level for exceptions raised within the unit and the test team should test it again during integration testing if it is designed to handle propagating exceptions.

Remember that functional testing will also cause many exceptions and uncover errors in the exception handlers. Thorough functional testing is a great aid in testing exception handlers, especially exception handlers that handle propagating exceptions.

Following is a summary of the guidelines for the structural testing of exception handlers at the unit testing level where propagating exceptions are not an issue.

(1) Test the exception handler using a structural testing metric. This guideline will be satisfied by meeting the statement coverage metric for the exception handler.

(2) Execute each RAISE statement that invokes the exception handler even if several raise the same exception. This guideline will be satisfied by meeting the statement coverage metric for the routine containing the exception handler.

(3) Determine if the exception handler is defined for a predefined exception(s). If so, do not attempt to raise each possible exception that will cause a particular exception handling routine to be executed. Raise only as many predefined exceptions as necessary for the exception handling routine to meet a structural testing metric.

(4) Execute RAISE statements that result in the propagation of an exception out of the routine being unit tested. There is no exception handler for the exception and the program will fail, but the statements must be executed to show that they can be reached.

Following is a summary of the guidelines for the structural testing of exception handlers at the integration testing level. Propagating exceptions to the exception handler are raised from lower level routines.

(1) When a thread is integrated, determine if exception handlers within that thread are designed to handle propagating exceptions. If all lower level routines in the calling hierarchy of the routine containing the exception handler are present (not stubbed) then the exception handler is ready for the remaining guidelines. If all lower level routines are not yet present, do not test the exception handler at this time. Test it at a later time when all lower level routines in the calling hierarchy are present. Remember to see if exception handlers, not previously ready, are now ready for testing. An exception handler will be ready for testing if all lower level routines are now present.

(2) Test only propagating exceptions that the exception handler has been designed to handle.

(3) Determine if the exception handler is designed for a user-defined exception(s). If so, identify each RAISE statement in lower level routines that cause an user-defined exception to be propagated to the exception handler. Derive a test case for each RAISE statement that will cause it to be executed. Execute each test case.

(4) Determine if the exception handler is designed for a predefined exception(s). If so, identify each lower level routine that can propagate a predefined exception to the exception handler. Identify a statement within each lower level routine that can raise the desired exception. Derive a test case for each statement that will cause the statement to raise the desired exception. Execute each test case. If it is not practical (too difficult and time-consuming) to raise an exception from a particular lower level routine, then do not do it.

4.5. Tasks

Tasks present many problems/error possibilities for the tester as noted in section 3.10 of Chapter 3. One of the problems concerns the structural testing of tasks.

The objective of the tester should be to execute all possible sequences of concurrency related events, which is in a sense path testing. This will uncover synchronization errors and deadlock. The problem for the tester is in determining all possible concurrency scenarios. If several tasks are involved, the number of possible scenarios can be unmanageable.

One way to help manage all the scenarios is through the use of automated tools. Research and development led by R. Taylor is ongoing at the University of California, at Irvine on the testing of tasking programs using automated tools [5]. The automated tools being developed employ static analysis and dynamic analysis techniques. These tools are part of several Ada software development tools that make up an APSE (Ada Programming Support Environment). At Cal-Irvine this environment is called Arcturus. It is not known when these tools will become commercially available.

The static analysis technique's objective is "to determine, for a given program, all possible sequences of concurrency events". [5] The tester will gain a knowledge of all synchronization that may occur during the execution of the program. Information derived from static analysis include the identification of all possible rendevous, detection of any deadlocks that may occur, and a listing of all parallel program activities.

There are problems with this technique. Determining all possible sequences of concurrency events can take a large amount of time. Static analysis is only accurate with static entities. The analysis is done independent of the actual execution environment. For example, the implications of DELAY statements are not taken into account. Finally, static analysis assumes that each intra-task path is executable.

The advantage of static analysis is that all possible sequences of concurrency related events are determined. This also helps in deriving test cases.

In dynamic analysis the Ada program A is transformed into another program A' (instrumented), so that A and A' have the same set of possible errors. During execution, A' will detect the imminency of deadlock, report the condition, and allow evasive action.

The dynamic analysis technique is not hindered by some of the problems associated with static analysis. The use of dynamic entities presents no problems and nearly all Ada constructs (i.e. DELAY statements) are taken into consideration.

Dynamic analysis has its problems, too. The error monitoring instrumentation can mean high overhead during execution. The presence of instrumented code can also disturb timing properties. "The overhead induced by the instrumentation may cause an observed phenomenon to disappear, though the potential for that error still remains" [5]. An example of this is shown in [27].

Another limitation is that error-free runs do not prove correctness of code.

Taylor [5] concludes the discussion by suggesting that static and dynamic analysis have complimentary characteristics. Thus, their joint use can counter each other's weaknesses. For example, static analysis results can help in reducing dynamic analysis overhead. If certain tasks have been tested to meet some criteria, then the instrumentation used for their monitoring may be eliminated, reducing overhead.

Taylor states that "no claim is made that the techniques presented are adequate" [5]. If the techniques described above, which use automated tools, are not adequate in testing tasks, then the present Ada tester is at a real disadvantage. He does not have automated tools at his disposal and it is not known when they will be at his disposal.

The testing guidelines which follow will compensate for the lack of automated testing tools.

These structural testing guidelines for tasks are given for unit testing and integration testing. The developer of the task performs unit testing and a test team performs integration testing.

During unit testing execute each entry call and ACCEPT statement in the task body one time. Use stubs/drivers to provide proper responses. This will be satisfied by meeting the statement coverage metric.

Determine if an array of tasks is declared or if the task can be created dynamically. Both the array of tasks and dynamic tasks are created through the use of a task type. A task type defines a template that may be used to create multiple tasks of the same type (same specification and body).

An example of an array of tasks is shown in Figure 14. Task SECURITY is a type. Any number of SECURITY tasks can be created. In this example, ALARM is an array of 50 tasks of type SECURITY, each with entries called ON_ALARM and OFF_ALARM. The ALARM array statement is the declaration that activates the alarm system. This, a collection of 50 tasks have been created and activated. Entry calls to these tasks look like this:

```
ALARM(33).ON_ALARM
ALARM(8).OFF_ALARM
```

```
task type SECURITY is
  entry ON_ALARM;
  entry OFF_ALARM;
end SECURITY;

ALARM : array(1..50) of SECURITY;
....
type SOUND is access SECURITY;
ALARM_X : SOUND;
....
ALARM_X := new SECURITY;
```

Figure 14. Creating Multiple Tasks.

The task type is also useful when it is not known how many tasks there will eventually be or if task identities need to be changed. This is called dynamic tasking. It is done by defining an access type to a task type. An example of this is shown in Figure 14. Type SOUND is an access type. A new task is activated by the "ALARM_X := new SOUND;" statement. This one task now has entry calls that look like this:

```
ALARM_X.ON_ALARM
ALARM_X.OFF_ALARM
```

It is recommended that if an array of tasks is declared, then create the maximum number of tasks when unit testing the task. Have each task, via a driver, accept each entry call one time. An entry call corresponds to an entry in the task specification.

If the maximum number of tasks cannot be determined because 1) an upper array bound is not declared and 2) the program specifications document does not indicate a maximum number, then create more than one task. Two is sufficient. Have each task, via a driver, accept each entry call one time.

It is recommended that if dynamic tasking is possible that the tester check the program specification document to see if there is a maximum number of dynamic tasks given for the program. If there is a maximum number, then create that maximum number of tasks during execution. If a maximum number cannot be determined, then create more than one task. Again, two is sufficient. For each task created, have each entry call defined in the task specification be accepted, via a test driver, one time.

Determine if a "family of entries" is declared for the task. This defines a set of peer entries. Each entry is indexed by a discrete value much like an array index. Execute each entry call, via a driver, one time. For example, see Figure 15. One task, COMM, is declared having a set of different entries. These entries can be referred to as:

```
COMM.SEND(HIGH)(LETTER)
COMM.RECEIVE(LOW)(LETTER)
```

In this example there are six possible entry calls.

Execute each one time.

```
type PRIORITY is (LOW,MEDIUM,HIGH);
task COMM is
  entry RECEIVE(PRIORITY)(X : out C_TYPE);
  entry SEND(PRIORITY)(X : in C_TYPE);
end COMM;
```

Figure 15. Family of Entries.

The integration testing of tasks is based on the pair-wise rendevous. A pair-wise rendevous involves an entry call in one task and a corresponding ACCEPT statement in another task. When they communicate, a rendevous has occurred.

The structural testing of rendevous is analogous to the structural testing of paths. Ideally, the testing goal for paths is to execute all logical paths in the program. Unfortunately, as described in section 2.6 of Chapter 2, this ideal cannot be reached because obtaining it involves exhaustive testing. So a more practical goal is to select a small but sufficient set of test paths to meet a structural testing metric.

The same holds true for rendevous. Ideally, the tester would like to execute all logical rendevous paths in the program. This ideal also cannot be reached because obtaining it involves exhaustive testing. So a more practical goal of executing all pair-wise rendevous one time is recommended.

This metric is met as follows. When a thread of tasks (one or more tasks) is integrated, the test team should identify all entry calls in the tasks. They should then identify all corresponding ACCEPT statements in another task for each entry call. Each of these pairs is a pair-wise rendevous.

For an example of determining pair-wise rendevous see Figure 16. Recall that a task reaching an entry call or ACCEPT statement may not proceed until a rendevous has been made. An exception to this rule is if a timeout condition is in effect. This information will help in determining pair-wise rendevous. In Figure 16 it appears that there are four pair-wise rendevous between the two tasks: 1-3, 1-4, 2-3, and 2-4. Actually, the 1-4 and 2-3 rendevous, assuming no timeout condition, will never occur because of the way Ada implements the rendevous mechanism. The test team should therefore execute the 1-3 and 2-4 rendevous one time.

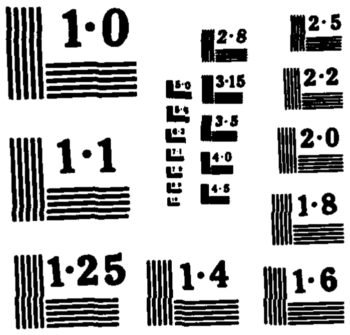
AD-A158 099 PRACTICAL GUIDELINES FOR TESTING ADA PROGRAMS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J B SNAUFER MAY 85 AFIT/CI/NR-85-58T 2/2

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

```
task TASK1;
task TASK2 is
  entry ACTIVITY;
end TASK2;
task body TASK1 is
  ....
  TASK2.ACTIVITY;  {1}
  ....
  TASK2.ACTIVITY;  {2}
  ....
end TASK1;
task body TASK2 is
  ....
  accept ACTIVITY;  {3}
  ....
  accept ACTIVITY;  {4}
  ....
end TASK2;
```

Figure 16. Task Rendevous.

If the task containing the ACCEPT statement has not yet been integrated the test team should take the following steps. Develop a stub to provide the proper response for the entry call during current testing. Annotate the entry call to make sure that it is executed later when the task containing the corresponding ACCEPT statement(s) is integrated. The ideal would be for all tasks to be integrated at the same time, but in large programs this is not a good integration testing strategy (big bang testing).

If the task containing the corresponding ACCEPT statements is present when the task containing the entry call is integrated, then derive test case(es) that will cause the rendevous. Then execute each test case.

When integration testing is complete, every entry call and ACCEPT statement should have been involved in the execution of one rendevous.

Following is a summary of the guidelines for the structural testing of tasks at the unit testing level.

(1) Test the task using a structural testing metric. This guideline will be satisfied by meeting the statement coverage metric.

(2) Determine if a "family of entries" is declared. If so, execute each entry call, via a driver, one time.

(3) Determine if the task can be defined as an array (collection) of tasks. If so, determine if there is a maximum number of tasks by looking at the array statement or by checking the program specification document. If the maximum number can be determined then create that number of tasks. If the maximum number cannot be determined, then create more than one task. Regardless of the number of tasks, have each task, via a driver, accept each entry call one time. An entry call corresponds to an entry in the task specification.

(4) Determine if the task can be dynamically created. If so, follow guideline 2 in this set of guidelines except for one statement. The determination of whether there is a maximum number of tasks cannot be determined by looking at an array statement. Refer to the program specification document.

Following is a summary of the guidelines for the structural testing of tasks at the integration level.

(1) Identify all entry calls in the thread of tasks being integrated. Identify all corresponding ACCEPT statement(s) for each entry call by looking at other tasks. This determines the pair-wise rendevous. If an identified ACCEPT statement is in a task that has not yet been integrated, then stub it until the task containing it is integrated. Make sure the rendevous is executed later when the task is integrated.

(2) If an identified ACCEPT statement is in a task that is present, then derive a test case that will cause the rendevous. Then execute the test case. This will meet the criteria of executing all pair-wise rendevous one time.

CHAPTER 5

CASE STUDIES

In this chapter, four case studies are presented to demonstrate the testing guidelines given in Chapter 4. Each case study corresponds to one of the features for which there are supplemental guidelines.

Each case study consists of a sample routine, sample test cases, and a step by step description of how to apply the supplemental guidelines of Chapter 4 when testing this routine. The step by step description is located in this chapter. The sample program and test cases are located in appendices.

Each numbered step for a feature in this chapter corresponds to the numbered guideline for that feature in chapter four. The same numbered step also corresponds to a numbered test case in the appropriate Appendix.

The test cases consist of functional test cases, Pascal-like language structural test cases, and test cases that result from following the supplemental guidelines. The functional and Pascal-like language structural test cases are given to show how the supplemental guidelines fit in to the general testing strategy. Assume that statement coverage is the structural testing metric.

A case study is presented for subprogram default parameters, generic program units, exception handling, and tasks.

5.1. Subprogram Default Parameters

The sample routine and test cases for this case study are located in Appendix A.

There are two default parameters in the COSTS procedure: QTY and UNIT_PRICE. The default value for QTY is 10 and default value for UNIT_PRICE is 10.00. Assume other routine(s) have done error checking on the data passed to the COSTS procedure. This results in perfect data being passed to COSTS. Assume the specification document has established a maximum discount of 50%.

The testing of COSTS in this case study occurs at the unit testing level. Functional and structural test cases are shown before the test cases that result from following the supplemental guidelines for subprogram default parameters.

The functional testing method utilized to derive the functional test cases is equivalence class testing. QTY is the variable that is broken into equivalence classes. The three classes are reflected in the case statement in

COSTS. The minimum (0%) and maximum (50%) discounts are also tested in the first and last functional test cases.

No structural testing test cases are needed because the statement coverage metric was met during functional testing.

Following is a step by step description of the unit testing supplemental guidelines for subprogram default parameters.

(1) This test case consists of all default values declared for COSTS. QTY equals 10 and UNIT_PRICE equals 10.00.

(2) These two test cases consist of one default value each. Test case 2a has QTY equal to 10 and test case 2b has UNIT_PRICE equal to 10.00.

(3) There are two default parameters. The criteria of four or less default values is met. However, no more test cases are necessary. The reason is that all combinations of the two default values have already been tested. Therefore, unit testing of COSTS is complete.

(4) There are not four or more default parameters. This guideline is ignored.

5.2. Generic Program Units

The sample routine and test cases for this case study are located in Appendix B.

One generic default parameter is defined. It is ORDER and its default value is ASCEND. Assume that the specifications document states that the maximum number of elements to be sorted is 500. Assume perfect data to be sorted is passed to the generic routine.

Following is a step by step description of the unit testing supplemental guidelines for the generic sample routine.

(1) Three types are identified in the generic part. Referring to Figure 11 in chapter four, the generic type parameters are numbers 2, 5, and 9. The actual data type for each in the first test instantiation is INTEGER. The test instantiation is created with no default value.

(2) An invocation is created. It is used to test the generic routine.

(3) First perform functional testing. Recall that even though functional testing is part of the general testing strategy, it cannot occur until the first two guidelines have been followed. Test case 3a contains

integers in ascending order while test case 3b contains integers in descending order. Test case 3c consists of integers that are in no particular order. Test cases 3d and 3e consist of two integers in different orders. Test case 3g consists of the maximum number of integers (500). Test case 3h consists of the minimum and maximum integer values allowed. No structural test cases are needed because the statement coverage metric was met during functional testing.

(4) A generic default value is defined. The parameter is ORDER and its default value is ASCEND. One test instantiation is created using the default value. Use some of the test cases that were derived for the first test instantiation. These are test cases 3a-3g. The other test case is a boundary test case and it needs to be executed only during the testing of the initial test instantiation. Use the invocation that was used to test the first instantiation (2) to test this test instantiation. There is just one default parameter, so no more test instantiations are necessary. The unit testing of this generic routine is complete.

The remaining guideline is for an instantiation that occurs during integration testing. Simply test each instantiation with functional test cases.

5.3. Exception Handling

The sample routine and test cases for this case study are located in Appendix C.

The procedure LEVEL_1 exception handler is the focus of this case study. Procedure LEVEL_2 is called by procedure LEVEL_1 and procedure LEVEL_3 is called by procedure LEVEL_2. Assume the lower bound of array TABLE is 1 and the upper bound is 100.

The testing of the procedure LEVEL_1 exception handler occurs first at the unit testing level. Assume that the structural testing metric statement coverage is for the main body (statements 1-20). Functional and structural test cases are shown before the test cases that result from following the supplemental guidelines for exception handlers.

The functional testing method utilized to derive the functional test cases is equivalence class testing. A is the variable that is broken into equivalence classes. The test cases a, b, and c are based on classes that are reflected in the case statement (statement 2). Assume that management has decided that equivalence class testing is sufficient for LEVEL_1.

After functional testing, LEVEL_1 statements 8,9,17, and 18 in the main body have not been executed. Structural test cases are needed. Test case d will cause the execution of statements 8 and 9. Test case e will cause the execution of statements 17 and 18. The statement coverage metric is now met for the main body of LEVEL_1.

Following is a step by step description of the unit testing supplemental guidelines for the LEVEL_1 exception handler.

(1) The statement coverage metric has not been met for the exception handler. Only statements 22-28 were executed during functional and structural testing. Statements 24-25 were executed by test case d as the L1_ERROR exception is raised at statement 9 because A equals 130. Statements 23 and 26-28 were executed by test case c as the L1_ERROR exception is raised at statement 14 because D equals 550. Supplemental test cases must be executed to raise exceptions that will cause the execution of the remaining exception handler statements. Test case 1a will raise a CONSTRAINT_ERROR exception which results in the execution of statements 29 and 30. The CONSTRAINT_ERROR exception is raised from statement 10 because A equals 0. Test case 1b will raise a NUMERIC_ERROR exception which results in the execution of

statements 31 and 32. The NUMERIC_ERROR exception is raised from statement 20 because A equals 43 and C equals 100. The remaining two statements, 33 and 34, are raised by a driver routine that contains the statement "raise L3_ERROR". This satisfies the statement coverage metric for the exception handler routine.

(2) Each explicit RAISE statement that invokes the exception handler has been executed. The RAISE statement of statement 9 was executed by test case d. The RAISE statement of statement 14 was executed by test case c.

(3) The LEVEL_1 exception handler is designed to handle the predefined exception CONSTRAINT_ERROR. The structural testing metric has already been met in the exception handling routine for a CONSTRAINT_ERROR by executing test case 1a. No more test cases are necessary.

(4) There are two RAISE statements that cause an exception to be propagated out of LEVEL_1. The RAISE statement at statement 18 was executed by test case e. The RAISE statement at statement 32 was executed by test case 1b. No more test cases are necessary.

This concludes the unit testing of the LEVEL_1 exception handler. Next is a description of test cases for the integration testing of the LEVEL_1 exception handler.

Use the same functional test cases, a-c, that were used during the unit testing of LEVEL_1.

Following is a step by step description of the integration testing guidelines for this thread. Assume that all three procedures in the sample routine are in the same thread.

(1) The LEVEL_1 exception handler is designed to handle two propagating exceptions: CONSTRAINT_ERROR and L3_ERROR. All lower level routines for LEVEL_1 are present.

(2) L3_ERROR and CONSTRAINT_ERROR are the propagating exceptions that the LEVEL-1 exception handler is designed to handle. A NUMERIC_ERROR exception will not be handled at LEVEL_1. It is propagated upward.

(3) The exception handler is designed for the user-defined exception L3_ERROR. That exception can be raised only from procedure LEVEL_3. A raise statement that can raise the exception is identified. It is at statement 53 in LEVEL_3. Test case 3 causes the RAISE statement to be executed.

APPENDIX C

Exception Handling Case Study

```
procedure TEST1 is new BUBBLE_SORT(ELEMENT => NUM,
                                   INDEX => MEMBERS,
                                   LIST => TABLE,
                                   ORDER => DESCEND);

(2) TEST1 (TEST_LIST);
```

Functional Test Cases

(3a)	-2 -1 0 1 2
(3b)	2 1 0 -1 -2
(3c)	4 -2 1 -5 3
(3d)	1
(3e)	1 -1
(3f)	-1 1
(3g)	A list of 500 integers
(3h)	-100 10 -10 100

Structural Test Cases

Structural metric met by functional test cases

Supplement Test Cases

```
(4) procedure T2 is new BUBBLE_SORT(ELEMENT => NUM,
                                    INDEX => MEMBERS,
                                    LIST => TABLE);
```

Functional Test Cases

Execute test cases 3a-3g from above.

SAMPLE ROUTINE

```
package GENERIC_SORT is
  type DIRECTION is (ASCEND,DESCEND);

generic
  type ELEMENT is private;
  type INDEX is range <>;
  type LIST is array (INDEX range <>) of ELEMENT;
  ORDER : in DIRECTION := ASCEND;
  with Function ">" (A,B : in ELEMENT)
       Return Boolean is <>;
  with Function "<" (A,B : in ELEMENT)
       Return Boolean is <>;
procedure BUBBLE_SORT (TABLE : in out LIST);
end GENERIC_SORT;

package body GENERIC_SORT is
procedure BUBBLE_SORT (TABLE : in out LIST) is
  LIMIT  : INDEX := TABLE'LAST;
  TEMP   : ELEMENT;
  SORTED : BOOLEAN;
begin
  SORT_LOOP;
    loop
      SORTED := TRUE;
      LIMIT := LIMIT + 1;
      for COUNT in 1..TABLE'LAST - 1
        loop
          if ((TABLE(COUNT) > TABLE(COUNT + 1)) and
               ORDER = ASCEND)                   or
             ((TABLE(COUNT) < TABLE(COUNT + 1)) and
               ORDER = DESCEND)                  then
            SORTED := FALSE;
            TEMP := TABLE(COUNT);
            TABLE(COUNT) := TABLE(COUNT + 1);
            TABLE(COUNT + 1) := TEMP;
          end if;
        end loop;
      exit SORT_LOOP when SORTED;
    end loop SORT_LOOP;
end BUBBLE_SORT;
end GENERIC_SORT;
```

SAMPLE TEST CASES

```
(1) using these types:
    type NUM  is range -100..100;
    type MEMBERS is range    1..500;
    type TABLE is array (MEMBERS range <>) of NUM;
```

APPENDIX B

Generic Program Unit Case Study

SAMPLE ROUTINE

```
procedure COSTS (QTY            : in INTEGER := 10;
                 UNIT_PRICE     : in REAL    := 10.00;
                 DISCOUNT_COST  : out REAL;
                 INITIAL_COST   : out REAL;
                 L_DISCOUNT     : in  REAL;
                 M_DISCOUNT     : in  REAL;
                 H_DISCOUNT     : in  REAL) is

begin
  INITIAL_COST := QTY * UNIT_PRICE;
  case QTY is
    when 1..5    => DISCOUNT_COST := INITIAL_COST *
                    (1.0 - L_DISCOUNT);
    when 6..25   => DISCOUNT_COST := INITIAL_COST *
                    (1.0 - M_DISCOUNT);
    when others => DISCOUNT_COST := INITIAL_COST *
                    (1.0 - H_DISCOUNT;
  end case;
end COSTS;
```

SAMPLE TEST CASES

Functional Test Cases

QTY	UNIT_PRICE	L_DISCOUNT	M_DISCOUNT	H_DISCOUNT
3	1.00	0.00	0.05	0.10
12	5.00	0.05	0.10	0.15
32	20.00	0.10	0.20	0.50

Structural Test Cases

Structural metric met by functional test cases

Supplement Test Cases

(1)	10	10.00	0.00	0.05	0.10
(2a)	10	5.00	0.05	0.10	0.20
(2b)	5	10.00	0.10	0.30	0.50

(3) Already Satisfied

(4) Not Applicable

APPENDIX A

Subprogram Default Parameter Case Study

[27] S. M. German, D. P. Helmbold, and D. C. Luckham, "Monitoring for deadlocks in Ada tasking," in _proc_. _AdaTEC_ _Conf_. _on_ _Ada_, Oct. 1982, pp. 10-25.

pp. 380-390, July 1982.

[14] L. White and G. Cohen, "A domain strategy for computer program testing," IEEE Trans. Software Eng., pp. 247-257, May 1980.

[15] G. J. Myers, The Art of Software Testing. New York: John Wiley and Sons, 1979.

[16] S. Rapps and E. Weyuker, "Data flow analysis techniques for test data selection," in proc. IEEE 6th Int. Conf. Software Eng., Sept. 1982, pp. 272-278.

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

[18] C. Gannon, "JAVS: A JOVIAL automated verification system," in Proc. IEEE COMPSAC 78, Nov. 1978, pp. 539-544.

[19] D. Casey and R. Erickson, "Practical tools for software test certification," in Proc. IEEE COMPCON 84, Feb. 1984, pp. 87-90.

[20] R. Taylor, "An integrated and verification and testing environment," Software Prac. and Exp., pp. 697-711, Aug. 1983.

[21] J. Arthur and J. Ramanathan, "Development of tools for selective program analysis," in Proc. IEEE COMPSAC 80, Oct. 1980, pp. 520-526.

[22] A. Feuer and N. Gehani, Comparing and Assessing Programming Languages Ada, C, and Pascal. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[23] M. R. Gardner, "Experiences in Ada software production," in Proc. IEEE COMPCON 83, Nov. 1983, pp. 404-407.

[24] A. R. Tripathi, W. D. Young, and D. I. Good, "A preliminary evaluation of verifiability in Ada," in Proc. ACM Annual Conf., Oct. 1980, pp. 218-224.

[25] N. G. Leveson and J. L. Stolzy, "Software fault tree analysis applied to Ada," in Proc. IEEE COMPSAC 84, Nov. 1984, pp. 458-467.

[26] S. T. Redwine, "An engineering approach to software test data design," IEEE Trans. Software Eng., pp. 191-200, March 1983.

BIBLIOGRAPHY

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

[2] J. Collofello and G. Klinkel, "An automated Pascal test coverage assessment tool," in Proc. IEEE COMPSAC 82, Nov. 1982, pp. 626-633.

[3] J. Collofello and A. Ferrara, "An automated Pascal multiple condition test coverage tool," in Proc. IEEE COMPSAC 84, Nov. 1984, pp. 20-26.

[4] G. Booch, Software Engineering With Ada. Menlo Park, CA: Benjamin/Cummings, 1983.

[5] R. N. Taylor and T. A. Standish, "Steps to an advanced Ada programming environment," in proc. IEEE 7th Int. Conf. Software Eng., Mar. 1984, pp. 116-125.

[6] I. C. Pyle, The Ada Programming Language. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[7] L. K. Belden, "Toward automatic testing of flight software," in proc. IEEE/AIAA 5th Digital Avionics Systems Conf., Nov. 1983, pp. 7.3.1-7.3.3.

[8] ----, Ada Reference Manual. U. S. Dept. of Defense, July 1980.

[9] R. Pressman, Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 1982.

[10] J. Goodenough and S. Gerhart, "Toward a theory of test data selection," IEEE Trans. Software Eng., pp. 156-173, June 1975.

[11] K. Foster, "Error sensitive test cases analysis (ESTCA)," IEEE Trans. Software Eng., pp. 258-264, May 1980.

[12] W. Howden, "Weak mutation testing and completeness of test sets," IEEE Trans. Software Eng., pp. 371-379, July 1982.

[13] L. Clark, J. Hassell, and D. Richardson, "A close look at domain testing," IEEE Trans. Software Eng.,

tain Ada features. They are to supplement the general testing strategy. The additional guidelines were given for subprogram default parameters, generic program units, exception handlers, and tasks.

Finally, case studies were provided which demonstrate the general testing strategy and the supplemental guidelines. Each case study consisted of a sample routine, sample test cases, and a step by step description of how to apply the supplemental guidelines to the sample routine.

This thesis provides an early look at testing Ada programs. Much more research needs to be conducted. Following is a list of possible future research areas.

(1) Automated test tools need to be researched and developed for the structural testing of Ada programs. Specifically, tools need to be developed to support the guidelines given in this thesis. The only presently known effort in this area is occurring at Cal-Irvine led by R. Taylor.

(2) The effectiveness of the guidelines presented in this thesis needs to be evaluated. Are they cost-effective? Are they adequate? Do they need to be refined? These are some of the questions that need to be investigated regarding this thesis.

CHAPTER 6

CONCLUSION

Practical testing guidelines have been given for Ada programs.

A general testing background of testing was presented. This background included a description of functional testing methods, Pascal-like language structural testing methods, and testing steps. These methods and steps are the foundation of the general testing strategy for Ada programs.

Ada was then analyzed from a testing point of view. Ada features were identified that are different from Pascal-like languages. They were then analyzed to determine if Pascal-like language structural testing methods are adequate for the structural testing of the feature. The feature was briefly described and problems/errors that might occur when using the feature were described. If a new structural testing approach was needed, then testing guidelines were provided for that feature in Chapter 4.

A general testing strategy was then presented. It detailed how Ada should be tested as a whole. Then additional structural testing guidelines were given for certesting.

Use the same functional test cases, a and b, that were used during the unit testing of PLAYER. Also execute test case c to check the efficiency/performance of the tasks.

Following is a step by step description of the integration testing supplemental guidelines for this thread. Assume that all 3 tasks in the sample routine are in the same thread.

(1) All entry calls are identified. They are: statement b in scorer; statement d in DEALER; and statements g and h in PLAYER. All ACCEPT statements for each entry call are identified. There are four pair-wise rendevous. They are: b-e between SCORER and DEALER; d-f between DEALER and PLAYER; and g-a and h-c between PLAYER and SCORER. All ACCEPT statements are in tasks that have been integrated.

(2) Test case a or b will cause each rendevous to be executed. This meets the criteria of executing each pair-wise rendevous once.

The functional testing method is very simple. There is no input to the task. There is a certain number of players (PLAYER tasks) that is determined by the type TABLE declaration. Test case a is for one PLAYER task. There is no maximum number of players that can play so test case b is for two players.

No structural testing test cases are needed because the statement coverage metric was met during functional testing.

Following is a step by step description of the unit testing supplemental guidelines for the PLAYER task.

(1) The PLAYER task has already met the structural testing metric. This was met by the functional test cases.

(2) A family of entries is not declared.

(3) The tasks can be defined as an array of tasks. The maximum number of tasks cannot be determined. Therefore, two tasks should be created. This has already been done during functional testing via test case b.

(4) The task cannot be dynamically created.

This concludes the unit testing of the task PLAYER. Next is a description of the test cases for integration

(4) The exception handler is designed for the predefined exception CONSTRAINT_ERROR. Two lower level routines, LEVEL_2 and LEVEL_3, can propagate a CONSTRAINT_ERROR exception to LEVEL_1. One statement is ident·:ied in each of the lower level procedures that causes the CONSTRAINT_ERROR exception to be raised to LEVEL_1. In LEVEL_2 the statement is number 42. Test case 4a will cause it execution. I will equal 114. In LEVEL_3 the statement is number 54. Test case 4b will cause ts execution. Y will equal 101.

5.4. Tasks

The sample routine and test cases for this case study are located in Appendix D.

Three tasks are defined in procedure RACE. One, PLAYER, is a family of tasks. It is the focus of this case study. Any number of players (task PLAYER) can take part in the race. The other two tasks are SCORER and DEALER. Each entry call/ACCEPT statement in the tasks are lettered. This will help show the pair-wise rendevous.

The testing of the task PLAYER occurs first at the unit testing level. Functional and structural test cases are shown before the test cases that result from following the supplemental guidelines for tasks.

SAMPLE ROUTINE

```
procedure LEVEL_1(A : in out INTEGER;
                  B : in out INTEGER;
                  C : in out INTEGER;
                  D : in out INTEGER;
                  TABLE : in out X_ARRAY);

procedure LEVEL_2(F : in out INTEGER;
                  G : in out INTEGER;
                  GROUP : in out X_ARRAY);

procedure LEVEL_3(Y : in out INTEGER;
                  LIST : in out X_ARRAY);

procedure LEVEL_1(A : in out INTEGER;
                  B : in out INTEGER;
                  C : in out INTEGER;
                  D : in out INTEGER;
                  TABLE : in out X_ARRAY) is

  L1_ERROR    : exception;
  L3_ERROR    : exception;
  MAX_A       : INTEGER := 100;
  MAX_D       : INTEGER := 500;

1 begin {LEVEL_1}
2   case A is
3     when 1..5   => A := A * 3;
4     when 6..30  => A := A - 15;
5     when others => A := A + 10;
6   end case;
    begin {block for exception handler}
7   if A > MAX_A
8     then
9       raise L1_ERROR;
    end if;
10  TABLE(A) := C + D;
11  D := D + 50;
12  if D > MAX_D
13    then
14      raise L1_ERROR;
    end if;
15  C := A * B;
16  if C = 50
17    then
18      raise C_ERROR;
    end if;
19  LEVEL_2(A,B,TABLE);
20  B := A ** C;
```

```
21  exception
22    when L1_ERROR =>
23      if A > MAX_A
24        then
25          A := 10;
        end if;
26      if D > MAX_D
27        then
28          D := 15;
        end if;
29    when CONSTRAINT_ERROR =>
30      A := 1;
31    when NUMERIC_ERROR =>
32      raise;
33    when L3_ERROR =>
34      A := 3;
35  end; {block for exception handler}
36 end LEVEL_1;


  procedure LEVEL_2(F : in out INTEGER;
                    G : in out INTEGER;
                    GROUP : in out X_ARRAY) is

    L2_ERROR  : exception;
    MAX_VALUE : INTEGER := 150;
    I         : INTEGER;

    begin {LEVEL_2}
37    F := F + 2;
      begin {block for exception handler}
38    if F + G > MAX_VALUE
39      then
40        raise L2_ERROR;
      end if;
41    I := F + G;
42    GROUP(I) := F * G + 1;
43    LEVEL_3(F,GROUP);
44    exception
45      when L2_ERROR =>
46        F := 5;
47        G := 10;
48    end; {block for exception handler}
49  end LEVEL_2;

    procedure LEVEL_3(Y : in out INTEGER;
                      LIST : in out X_ARRAY) is

50    begin {LEVEL_3}
51      if Y > MAX_A + 1
52        then
```

```
53          raise L3_ERROR;
          end if;
54        LIST(Y) := Y -3;
55      end LEVEL_3;
```

SAMPLE TEST CASES

UNIT TESTING

Functional Test Cases

	A	B	C	D
(a)	5	20	10	150
(b)	20	7	5	10
(c)	32	2	12	500

Structural Test Cases

(d)	120	3	2	1
(e)	25	5	3	3

Supplement Test Cases

(1a)	15	3	3	3
(1b)	33	1	100	2

(1c) Driver to raise L3_ERROR.

(2) Already Satisfied

(3) Already Satisfied

(4) Already Satisfied

INTEGRATION TEST CASES

Functional Test Cases

Execute test cases a-c from above

Supplement Test Cases

	A	B	C	D
(1)	No test cases produced			
(2)	No test cases produced			
(3)	90	3	2	2
(4a)	4	90	10	10
(4b)	89	13	6	7

APPENDIX D

Tasks Case Study

SAMPLE ROUTINE

```
with TEXT_IO; use TEXT_IO;

procedure RACE is

  type MEMBERS is INTEGER range 1..1000;

  task SCORER is
    entry ADD(MEM : in MEMBERS;
              TOT : in out INTEGER);
    entry SUB(TOT : in out INTEGER);
  end SCORER;

  task DEALER is
    entry WINNER(MEM : in MEMBERS;
                 TOT : in INTEGER);
  end DEALER;

  task type PLAYER is
    entry ID(MEM : in MEMBERS);
  end PLAYER;

  type TABLE is array (MEMBERS) of PLAYER;
  GROUP : TABLE;
  GAME  : TEXT_IO.FILE_TYPE;

  task body SCORER is
    begin {SCORER}
      loop
        select
(a)       accept ADD(MEM : in MEMBERS;
                     TOT : in out INTEGER) do
            TOT := TOT + 2;
            if TOT > 25
              then
(b)             DEALER.WINNER(MEM,TOT);
            end if;
          end ADD;
        or
(c)       accept SUB(TOT : in out INTEGER) do
            TOT := TOT -1;
          end SUB;
        end select;
      end loop;
    end SCORER;

  task body DEALER is

    ID : MEMBERS;
  begin {DEALER}
    for INDEX in MEMBERS
      loop
(d)     GROUP(INDEX).ID(INDEX);
      end loop;
(e) accept WINNER(MEM : in MEMBERS;
                  TOT : in INTEGER) do
      put(GAME,"The winner is ");
      put(GAME,MEM,4);
      put(GAME," with a score of ");
      put(GAME,TOT,2);
    end WINNER;
  end DEALER:

  task body PLAYER is

    IDENTITY : MEMBERS;
    SCORE    : INTEGER := 0;

  begin {PLAYER}
(f) accept ID(MEM : in MEMBERS) do
      IDENTITY := MEM;
    end ID;
    loop
(g)   SCORER.ADD(IDENTITY,SCORE);
      delay 0.25;
(h)   SCORER.SUB(SCORE);
      delay 0.25;
    end loop;
  end PLAYER;

begin {RACE}
  create (GAME,OUT_FILE,"race.out");
end RACE;
```

<u>SAMPLE TEST CASES</u>

<u>UNIT TESTING</u>

Functional Test Cases

(a) type TABLE is array (1..1) of PLAYER;

(b) type TABLE is array (1..2) of PLAYER;

Structural Test Cases

Structural metric met by functional test cases

Supplement Test Cases

(1) Already Satisfied

(2) Not Applicable

(3) Already Satisfied

(4) Not Applicable

INTEGRATION TESTING

Functional Test Cases

Execute functional test cases a and b from above.

(c) type TABLE is array (MEMBERS) of PLAYER;

Supplement Test Cases

(1) No test cases produced

(2) Already Satisfied

BIOGRAPHICAL SKETCH

Joseph Blair Snaufer was born in Springfield, Ohio, on December 13, 1956. He received his elementary and secondary education in the New Carlisle-Bethel Public Schools. In 1975 he entered Bowling Green State University, graduating in 1979 with a Bachelor of Science degree in Computer Science. He was commissioned a Second Lieutenant in the United States Air Force and served at the Foreign Technology Division, Wright-Patterson Air Force Base, Ohio, from 1979 to 1983. In June, 1983, he entered Arizona State University, sponsored by the Air Force Institute of Technology, to study for a Master of Science degree in Computer Science. He currently holds the rank of Captain in the Air Force. He is a member of Alpha Lambda Delta and Upsilon Pi Epsilon, national honor societies. He is married and the father of one son.