Automating Software Module Testing for FAA Certification

Usha Santhanam
The Boeing Company
Wichita Maintenance and Modification Center
Wichita Kansas
1-316-526-4474
usha.santhanam@boeing.com

1. ABSTRACT

Structural testing, also called code-based or white-box testing, is performed for the purpose of exercising the code thoroughly. The code coverage measurements are used to evaluate the effectiveness of the test cases toward this goal. Manual, code-based testing is cumbersome and time-consuming. Automation provides help in selecting test data, applying those test cases to the software, and deciding whether a program has been tested enough by providing coverage metrics. This paper describes an automated module testing process for software written in Ada using a tool set called Test Set Editor (TSE). TSE consists of a proprietary set of programs which work in combination with Microsoft Excel spreadsheet program and home grown test scripts written in Tcl/tk. Use of the tool set is shown to yield significant cost savings in constructing structural tests needed to satisfy FAA certification requirements for safety-critical software.

1.1 Keywords

Software Module Test, Structural Test, Test Automation, FAA Certification.

2. INTRODUCTION

Automatic software testing is gradually becoming accepted practice in the software industry. The shrinking development cycle and higher expectation of software quality are forcing software teams to seek help from automation. Current state-of-the-practice in software test automation uses tool-specific programming and scripting languages to automate testing. Automation aims to achieve both reduced costs and a more thorough analysis, testing and verification and is vital to keeping pace with increasing software complexity.

2.1 Current Drawbacks in Code-Based Software Testing

Software complexity and accelerated development schedules make avoiding defects difficult. The user reported bugs occur in software because the user executed untested code, or the order in which the statements were executed in actual use differed from that during testing, or the user applied a combination of untested input values, or the user's operating environment was never tested [3]. About 80 percent of the defects in software come from 20 percent of the modules, and about half the modules are defect free. In addition, 90 percent of the downtime comes from, at most, 10 percent of the defects [2]. One of the common experiences during software testing is the difficulty of accurately and efficiently capturing and analyzing the sequence of events that occur when a program executes[4].

2.2 Tools and Testing Process

The automation techniques and tools adopted in our process are designed to achieve shorter turnaround time and cut cost during regression testing. Although commercial off-the-shelf test automation tools such as AdaTEST are available, either they are limited to a specific application domain or they require the tester to code the test cases in the form of a test driver function in some programming language. The test tools and process we describe are domain-independent and require only that the tester create test cases in a tab-delimited text file, perhaps using a spreadsheet program such as Excel. The rest of the process of testing is automated, removing the language-dependent test driver generation chores from the hands of the tester. A tester with limited knowledge of Ada is able to create test cases without having to worry about debugging test drivers.

Our tool set, called Test Set Editor (TSE), is a multi-purpose test tool set designed for use with Ada. It consists of a tool to identify the inputs and outputs of the module to be tested, a tool for building a driver program to apply a set of test cases to the module, and a tool for measuring coverage. Test cases are not automatically derived from code, but a test engineer must construct them. They may be derived from requirements or solely from code in order to attain coverage.

3. FAA CERTIFICATION TESTING

This brief overview of FAA certification testing is intended to put our tools and process in context for the reader; for a more complete treatment of the topic the reader should refer to RTCA DO-178B [6].

Testing safety-critical software begins with a set of requirements-based test cases. Ideally, these test cases should also achieve structural coverage; that is, the source code should be exercised completely. In practice, however, structural coverage is often not achieved through requirements-based testing alone, the most common reason being that it is impossible or impractical to achieve the combination of conditions that it takes to exercise every branch of a complex module from top-level inputs. When this happens, testing each module (subprogram) in isolation leads to the desired coverage.

There are three levels of structural coverage to deal with:

(a) statement coverage

(b) decision coverage

(c) modified condition-decision coverage (MCDC)

Statement coverage refers to executing each source-level statement at least once. Decision coverage, sometimes called branch coverage, refers to taking each execution branch at least once. Decision coverage goes beyond statement coverage; for example, an IF statement without an ELSE clause can be covered at statement level by a single test case, but decision coverage will require two test cases.

MCDC is a unique form of coverage that is designed to demonstrate the independence of individual conditions that constitute a complex decision. For example, if (A OR B) AND C is a decision, MCDC requires that each of the conditions A, B, and C be demonstrated to independently influence the outcome of the decision. That is, each condition can be toggled to vary the outcome while holding all other conditions constant. For an extensive discussion of MCDC, the reader is referred to [7].

One reason why module testing is suited for achieving structural coverage is that the tester is able to control the inputs to the module right at its doorsteps. The second reason is when the module under test calls another module, we can "stub" the called module and return arbitrary results from the stub in order to force downstream coverage in the calling module. This is quite acceptable as our goal is to verify that the module under test functions properly regardless of what its subordinate modules compute.

4. TSE TOOL SET AND PROCESS

The tasks of testing a module for structural coverage in the FAA certification context can be summarized as:

(a) identifying inputs and outputs of a module (or a collection of modules),

(b) selecting test cases and tagging them with requirements,

(c) constructing a test driver and stubs for subprograms called by the module under test,

(d) compiling and running the tests,

(e) analyzing coverage .

TSE assists the tester with all of these tasks except the selection of the test cases. We illustrate the tools and the process of testing a module using a simple example.

4.1 Example

Consider module testing procedure Get_End in package Io:

```
package Io is
  Debugging : Boolean := False;
  procedure Get_End;
end Io;

with Text_Io;
package body Io is
  K : constant := 1024;
  Max_Line_Length : constant := 16*K;
  Line : String (1 .. Max_Line_Length);

  procedure Error (Msg : String) is
  begin
    null;
  end Error;

  procedure Get_Line is
  begin
    null;
  end Get_Line;

  procedure Get_End is
  begin
    if Line (1 .. 4) = ".end" then
      Get_Line;
    else
      Error ("Missing .END, assume here.");
    end if;
  end Get_End;
end Io;
```

4.1.1 Step (a) Identifying inputs, outputs, and stubs

An examination of procedure Get_End shows that the global variable Line is an input to the module and that there are no outputs from the module. There are two procedures called from Get_End, namely Error and Get_Line. We will assume that these procedures are to be stubbed for this test.

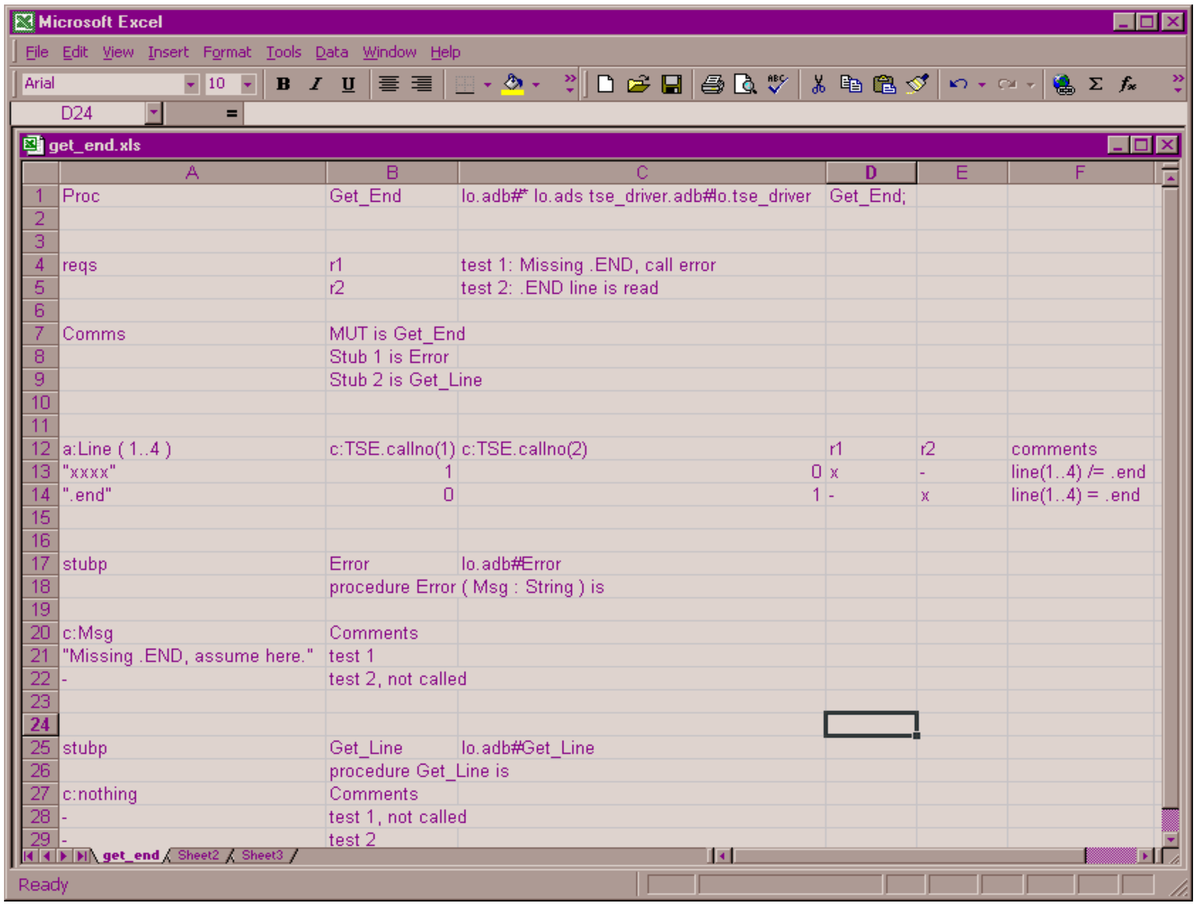
For a more complex procedure, it may be necessary to use the tool TemGen in our TSE tool set to determine all inputs, outputs, and called subprograms (potential stubs).

4.1.2 Step (b) Selecting test cases

In order to achieve statement and decision coverage, we will need two test cases, which are represented as requirements r1 and r2. Requirement r1 corresponds to the error case, and r2 to the normal case where a ".end" line is read.

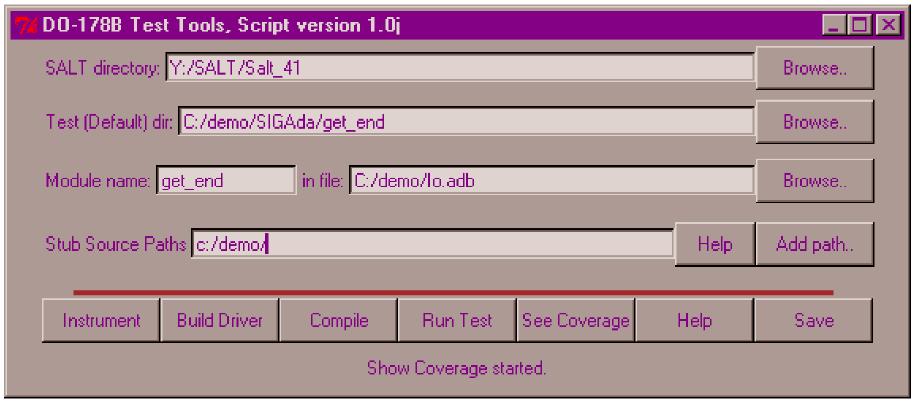
Figure 1 shows how all of this information is entered in a spreadsheet. The significant rows in this spreadsheet are the test data rows 13-14, which define values for the input variables (Line (1..4)), and check the expected results (number of calls to each of the two stubs). Columns D and E of these rows identify the requirements covered by each test case. Column F contains a comment that is echoed back at the output when the test is run.

Rows 21-22 represent checks made in stub Error; they correspond in order to the two test cases represented by rows 13-14. Likewise, rows 28-29 represent stub Get_Line.



	A	B	C	D	E	F
1	Proc	Get_End	lo.adb#* lo.ads tse_driver.adb#lo.tse_driver	Get_End;		
2						
3						
4	reqs	r1	test 1: Missing .END, call error			
5		r2	test 2: .END line is read			
6						
7	Comms	MUT is Get_End				
8		Stub 1 is Error				
9		Stub 2 is Get_Line				
10						
11						
12	a:Line (1..4)	c:TSE.callno(1)	c:TSE.callno(2)	r1	r2	comments
13	"xxxx"	1	0	x	-	line(1..4) /= .end
14	".end"	0	1	-	x	line(1..4) = .end
15						
16						
17	stubp	Error	lo.adb#Error			
18		procedure Error (Msg : String) is				
19						
20	c:Msg	Comments				
21	"Missing .END, assume here."	test 1				
22	-	test 2, not called				
23						
24						
25	stubp	Get_Line	lo.adb#Get_Line			
26		procedure Get_Line is				
27	c:nothing	Comments				
28	-	test 1, not called				
29	-	test 2				

Figure 1. Get_End.xls



SALT directory: Y:/SALT/Salt_41

Test (Default) dir: C:/demo/SIGAda/get_end

Module name: get_end in file: C:/demo/lo.adb

Stub Source Paths c:/demo/

Show Coverage started.

Figure 2. Tse_Tools.tcl

4.1.3 Step (c) Constructing test driver and stubs

Once the test cases have been entered into the spreadsheet, the information is saved as a tab-delimited text file. This file is used by Build Driver step (figure 2) to automatically generate the test driver and stub code, portions of which are shown below.

The test driver code:

```
procedure TSE_Driver is

  Excp_In_Mut : Boolean;
  Excp_Others : Boolean;

  package TSE renames TSE_Support;

begin

  -- Requirements table
    --  1 r1   test 1: Missing .END, call error
    --  2 r2   test 2: .END line is read

  -- Table of checks..
  -- Chk# : Description
    --  1 : Unexpected value (TSE.callno(1))
    --  2 : Unexpected value (TSE.callno(2))
    --  3 : Excp check (others) failed

  TSE.Init_Session (Module_Id =>  3890589);

  TSE.UserLog ("MUT is Get_End");
  TSE.UserLog ("Stub 1 is Error");
  TSE.UserLog ("Stub 2 is Get_Line");
  -- No user inits

  -- Test Case  1 --------------- Test Case  1 --
  -- line(1..4) /= .end
  --
  TSE.Init (Test => 1);
  -- Initialize inputs
  Line ( 1..4 ) := "xxxx";

  -- Call module under test --------------------
  begin -- excp handler wrapper
    Excp_Others := False;
    Excp_In_Mut := True;
    begin
      Get_End;
      Excp_In_Mut := False; -- no excp from MUT
      -- Check inputs are unchanged,
      -- outputs are as expected
      if (TSE.callno(1) /= 1) then
        TSE.Fail (Check => 1);
      end if;
      if (TSE.callno(2) /= 0) then
        TSE.Fail (Check => 2);
      end if;
    end; -- test case 1
  exception
  when Excp_Name: others =>
    Excp_Others := Excp_In_Mut;
    TSE.Save_Excp_Info
(Ada.Exceptions.Exception_Information
(Excp_Name));
  end; -- excp handler wrapper
  if Excp_Others then
    TSE.Fail (Check => 3);
    TSE.Show_Excp_Info;
  end if;
  -- Set covered requirements
  TSE.Covered (ReqNo => 1);
  TSE.Done (Test => 1);
        . . .
end tse_driver;
```

The code for stub Error:

```
procedure Error ( Msg : String ) is
  package TSE renames TSE_Support;
begin

  -- Table of checks..
  -- Chk# : Description
    --  8 : Error.Unexpected call to stub
    --  9 : Error.Unexpected value (Msg)

  TSE.Call_Stub (1);

  case TSE.TestNo is

  when  1 => -------------------- Test case  1 --
    -- test 1
    -- Check stub inputs
    if (Msg /= "Missing .END, assume here.") then
      TSE.Fail (Check => 9);
    end if;
    -- Set stub outputs

  when  2 => -------------------- Test case  2 --
    -- test 2, not called
    -- Check stub inputs
    -- Set stub outputs
    null;

  when others => -- unexpected test case number
    TSE.Fail (Check => 8);
  end case; -- TSE.TestNo
end; -- Error
```

4.1.4 Step (d) Compiling and running test

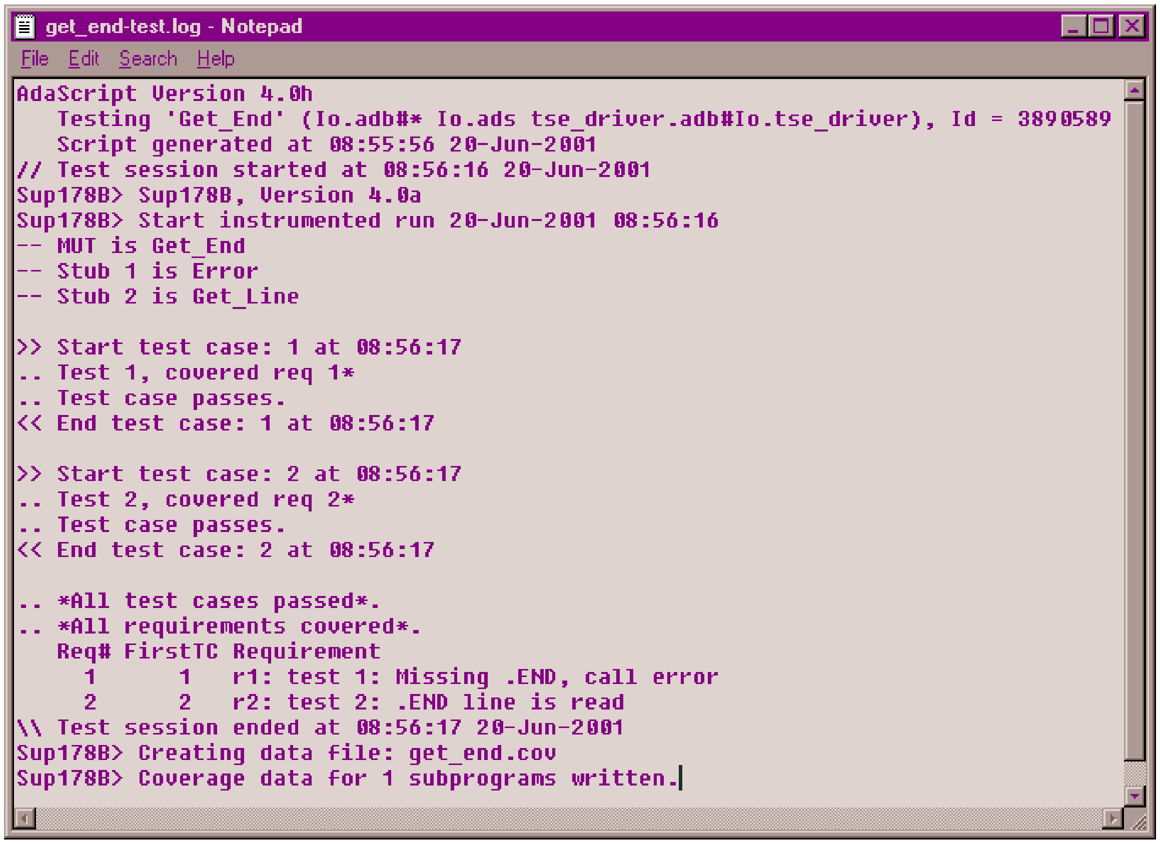
Once the code for the test driver and stubs is generated and inserted into the appropriate places (as indicated in the spreadsheet), the program is ready to be compiled (Compile in figure 2). When compilation is completed successfully, the test can be run (Run Test in figure 2), which yields the result shown in figure 3.

4.1.5 Step (e) Analyzing for coverage

If we wish to see the level of coverage achieved by our test cases, we need to begin by instrumenting the code (Instrument in figure 2). During the instrumentation process, the code for the module under test is modified to insert probes which will collect the necessary data during test run. Upon completion of the test run, the data collected is written to a file, and can be displayed in a variety of ways (See Coverage in figure 2). A plain text version of this display is shown in figure 4.

4.2 Regression testing

Rerunning a module test when source code changes have occurred is a simple task under TSE. Since all test cases are self-checking (meaning, each test case checks its own expected output), regression testing consists of running the test files through the steps outlined above. In TSE, this is achieved through a batch file so that manual effort is minimized. If one or more tests fail, the log file generated identifies the tests which need to be examined and/or updated.



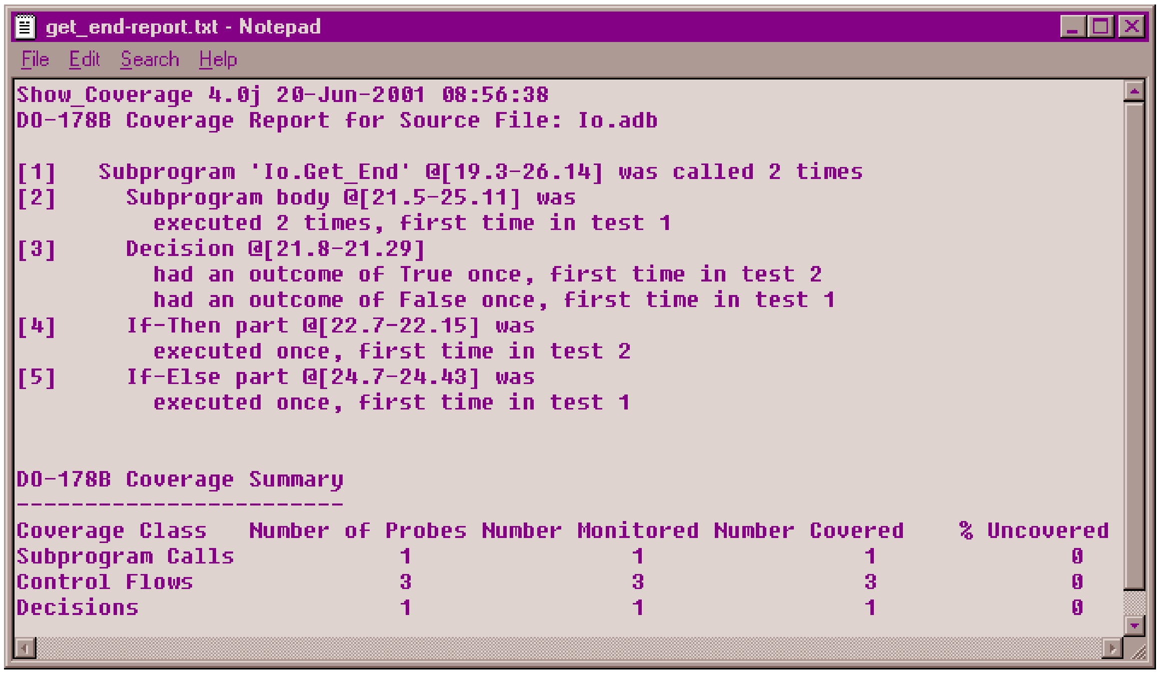
```
AdaScript Version 4.0h
   Testing 'Get_End' (Io.adb#* Io.ads tse_driver.adb#Io.tse_driver), Id = 3890589
   Script generated at 08:55:56 20-Jun-2001
// Test session started at 08:56:16 20-Jun-2001
Sup178B> Sup178B, Version 4.0a
Sup178B> Start instrumented run 20-Jun-2001 08:56:16
-- MUT is Get_End
-- Stub 1 is Error
-- Stub 2 is Get_Line

>> Start test case: 1 at 08:56:17
.. Test 1, covered req 1*
.. Test case passes.
<< End test case: 1 at 08:56:17

>> Start test case: 2 at 08:56:17
.. Test 2, covered req 2*
.. Test case passes.
<< End test case: 2 at 08:56:17

.. *All test cases passed*.
.. *All requirements covered*.
   Req# FirstTC Requirement
     1       1    r1: test 1: Missing .END, call error
     2       2    r2: test 2: .END line is read
\\ Test session ended at 08:56:17 20-Jun-2001
Sup178B> Creating data file: get_end.cov
Sup178B> Coverage data for 1 subprograms written.
```

Figure 3: Test Results



```
Show_Coverage 4.0j 20-Jun-2001 08:56:38
DO-178B Coverage Report for Source File: Io.adb

[1]   Subprogram 'Io.Get_End' @[19.3-26.14] was called 2 times
[2]     Subprogram body @[21.5-25.11] was
          executed 2 times, first time in test 1
[3]     Decision @[21.8-21.29]
          had an outcome of True once, first time in test 2
          had an outcome of False once, first time in test 1
[4]     If-Then part @[22.7-22.15] was
          executed once, first time in test 2
[5]     If-Else part @[24.7-24.43] was
          executed once, first time in test 1


DO-178B Coverage Summary
------------------------
Coverage Class   Number of Probes Number Monitored Number Covered   % Uncovered
Subprogram Calls           1                 1                1            0
Control Flows              3                 3                3            0
Decisions                  1                 1                1            0
```

Figure 4: Report

5. BENEFITS OF TSE

The benefits of TSE are significant:

- Tests can be developed by engineers who are not necessarily expert programmers in Ada. This is a huge advantage in an environment where Ada programmers are in short supply, especially if they must be brought in from outside for independence of verification and validation.
- TSE includes a tool to analyze the source code and identify inputs and outputs of a given module, including parameters and globally referenced variables. Additionally, all subprograms called by the module are also identified so that the tester may plan a stubbing strategy for them. This semantics-based analysis step ensures that inputs, outputs, and subprograms are not overlooked due to renaming or overloading.
- Coverage information is provided both in tabular form for certification documentation as well as in HTML form which can be browsed using standard browsers. This latter form is useful during development when the tester needs to know where coverage deficiencies exist.
- TSE tests are easy to maintain. Tool-generated driver and stubs have uniform structure that is simple to comprehend in the unlikely event that generated script must be examined. Tests written by one engineer can be modified or maintained by another years down the road.
- TSE tests are self-verifying. When the test is run, the output identifies the result as pass or fail. There is usually no need for a separate verification step.
- TSE tests are repeatable. Everything needed to run a test is captured in files; no manual interaction is normally needed. Testing for regression is easy, and can be totally automated.
- TSE tests can be administered on a host or the target platform. Using slightly different support package, the same tests can be run on either platform. TSE also provides the convenience of host platform for development, eliminates additional work to apply the tests on the target and even targets with no I/O capability can be handled.

5.1 Productivity Improvements

The TSE tool set has been used extensively in-house to develop both functional and structural tests. In order to measure productivity, the effectiveness of the TSE-toolset is compared with and without the use of the tools. A tool that numerically scores a module for testability with TSE is used to determine the effort that is required to complete the testing. The numerical score is a weighted sum of:

- statement count (sc),
- number of IF's (ic),
- CASE's (cc),
- LOOP's (lc),
- EXIT's (xc),
- logical operators AND, OR, XOR, and NOT (Logc),
- short-circuit control forms AND-THEN/OR-ELSE (Scc),
- stub calls (Stc):

Composite =

$$c + ic + 2*cc + 4*lc + xc + Logc + Scc + Stc*5$$

Our experience shows that the formula **Composite/2** is a good predictor for the number of hours needed to complete module testing by a moderately experienced test engineer. For Get_End the composite score is 14, and an estimated testing time of 14/2 = 7 hours.

For comparison, consider hand-coding the test driver and the stubs by an equally experienced software engineer. We will use a code productivity rate of 2 source lines of code per hour, which is a frequently used rule of thumb [5]. This rate is moderately aggressive, considering that the driver development effort must include test design and code documentation, both of which are an integral part of TSE-based testing. For Get_End, the test driver and stubs comprise of 90 SLOC, leading to an estimated test generation time of 90/2 = 45 hours. This suggests a 1:6.5 productivity advantage from using TSE over the manual approach for the module Get_End.

5.2 Limitations of TSE

Ada features such as generics and exception processing can be tested using the TSE tool set. Generics are handled on an instantiation by instantiation basis. That is, any given instantiation can be tested using TSE, but it is not possible to construct one generic module test to verify all instantiations. The module under test may raise exceptions. If propagated outside the module, the tool will handle such exceptions and report as expected or not expected. Object Oriented programs can also be tested using the TSE toolset, although as yet we have not done so.

Ada features such as tasking are difficult to test because tasks run in independent parallel threads. Our test driver is not suitable for testing an independent thread. However, each subprogram can be tested independently regardless of the number of tasks sharing it.

If a module uses private type parameters or a return private type, the construction of an external test driver is a bit awkward. However, it is possible to embed the test driver in the body of the package that defines the limited private type, which would allow us to treat the private types as non-private types and go around the limitation.

TSE can be used to test any library level subprogram or one that is embedded in a library-level package. At present, TSE cannot be used to test subprograms nested deeper as independent modules, as this would require embedding a driver in the same nested context as the subprogram to be tested, and providing handles at various levels to invoke that driver.

6. SUMMARY

Quality of software is becoming increasingly important and testing related issues are becoming crucial for ensuring high quality for safety critical software. Keeping this in mind we have designed our test automation to eliminate manual coding and minimize the need for programming knowledge. Our tool set has been shown to cut testing effort several-fold over the manual approach. Our process is also designed to be repeatable so that the cost of re-testing new versions of the software is kept to a minimum. This testing process has been used in the structural testing of FAA certified commercial avionics software as well as the qualification of tools used in the production of such software.

7. ACKNOWLEDGMENTS

I thank my manager Jerry White and my colleagues Ty Startzman and V. Santhanam for their review, comments, and suggestions.

8. REFERENCES

[1] Yamaura, T. How To Design Practical Test Cases, *IEEE Software*, November/December 1998, 30-36.

[2] Boehn, B., Basili, V.R. Software Defect Reduction Top 10 List. *Computer*, January 2001, 135-137.

[3] Whittaker, J.A. What Is Software Testing? And Why Is It So Hard? *IEEE Software*, January/February 2000, 70-79.

[4] Ball, T., Larus, J.R. Using Paths to Measure, Explain, and Enhance Program Behavior. *Computer*, July 2000, 57-65.

[5] Hihn, J., Habib-agahi, H. Cost estimation of software intensive projects: a survey of current practices. *Proceedings of the 13th international conference on Software engineering*, 1991, 276 – 287.

[6] *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, D.C., December 1992.

[7] Chilenski, J.J., Miller, S.P. Applicability of Modified Condition Decision Coverage to Software Testing. *Software Engineering Journal*, v.7, n.5, September 1994, 193-200.