A language for specifying software tests

by DAVID J. PANZL

General Electric Company
Schenectady, New York

INTRODUCTION

The execution of software test cases and the verification of test results may be performed automatically by a new type of program called an automatic software test driver. When using an automatic test driver, a formal test procedure is coded in a special test language. The test procedure takes the place of the test data and test setup instructions of conventional testing, and controls the automatic test driver. An automatic software test driver applies one test procedure to all or part of a target program, executes all of the test cases specified in the test procedure, and verifies that the results of each test case are correct. This paper describes the Fortran Test Procedure Language (TPL/F) which was developed at General Electric and is used for specifying test procedures for Fortran software.

The concept of automatic software test drivers is a new idea that has been evolving slowly over the past six years and just now seems to be approaching the point where soon it may play a significant role in the development and maintenance of production software. Two other automatic software test drivers that were developed independently in recent years and have their own software test languages are described in References 2 and 3.

The Fortran Test Procedure system is illustrated in Figure 1. One test procedure coded in TPL/F and the source code for one or more modules of the target program are processed by the automatic test driver which executes all of the test cases specified in the test procedure, and produces a brief test execution report (Figures 2 and 3) stating which test cases failed, if any, and the degree of testing coverage actually achieved by the test procedure.

Test cases consist of input data for the target program and model output data. The automatic test driver actually executes the target program for each test case, feeding the input data to the target program and comparing outputs from the target program with the model outputs specified in the test procedure. Incorrect outputs produce a diagnostic in the test execution report (Figure 3).

Since the TPL/F system has access to the target program’s source code, it can monitor which statements were actually executed and which branches were actually traversed while executing a test procedure. This type of information provides valuable feedback to the test designer and is generally not available when testing is done manually.

The specific goals of the TPL/F automatic software test driver are as follows. The need for writing drivers and stubs for module and subsystem testing is eliminated since the TPL/F system can test any combination of one or more modules independently of the rest of the target program. The TPL/F test language provides a standard format for specifying software tests and the test procedure processor provides a standard test execution setup. Since formal test procedures specify the correct outcomes of test cases, the test procedure processor automates the verification of test execution results.

TPL/F TEST PROCEDURES—AN OVERVIEW

TPL/F test procedures and the major elements of the TPL/F language are introduced in this section by means of an example. The following section describes the TPL/F language in greater detail. For the present example, we show a simple test procedure for the Fortran target module of Figure 4. The target module, IOSUB, is a subroutine and needs to be called by a higher level module for execution. IOSUB also invokes a lower level subroutine, LABEL, which is not under test and must be simulated when testing IOSUB. On each entry to IOSUB, one unformatted record is read from logical unit 12 and LABEL is called once.

A simple test procedure to test IOSUB is illustrated in Figure 5. The test procedure contains three test cases, each of which causes an actual execution of the target module, IOSUB. The MODULES statement names the target modules (IOSUB of Figure 4) under test.

TPL/F test cases are executed in three steps. First, initialization code is executed and puts the target program in a known initial state. In the example, the first two statements of each test case are used to initialize the target program. In the first test case, for example, they assign the values 1 and 0 to the variables N1 and N2, respectively, of IOSUB.

Second, the target program is executed. The form of the EXECUTE statement used in Figure 5 causes execution to begin at the first executable statement of IOSUB and terminate the first time a STOP or RETURN statement is encountered in IOSUB.
Third, after the test case execution has terminated, an assertion about the final state of IOSUB is verified. In the first test case of Figure 5, the variables N1 and N2 of IOSUB are verified to have the values of 6 and 0, respectively.

In TPL/F, stub versions of missing subprograms called by the target program are coded as Fortran-like subroutines and functions embedded in the test procedure. In Figure 5, for example, a stub version of subroutine LABEL begins at the SUBROUTINE statement and runs thru the following END statement. #TEST is an integer system variable whose value is always the index of the test case currently executing. The stub version of LABEL verifies that the argument passed by IOSUB when executing under the first test case is 10 and returns the value 0 for the argument on return. Should LABEL call IOSUB with an incorrect argument value, the ABORT statement would terminate the current test case (but not the test procedure) and put a diagnostic message in the test execution report.

The six statements beginning at the I/O SIMULATOR statement and continuing through the following END statement specify an I/O simulator for logical unit 12 of the target program. An I/O simulator can supply input data to the target program in response to READ statements or verify outputs from WRITE statements in the target program. The I/O simulator of Figure 5 specifies three unformatted records containing five integer values each, followed by an end-of-file mark. The first three records read by the target program from logical unit 12 will be taken from the I/O simulator. An attempt to read a fourth record would produce an EOF return in the target program.

Although the test procedure in this example contains only three test cases, most test procedures contain a much larger number. Typically, a test procedure for a single Fortran target module of 50 to 100 statements may contain 20 to 50 test cases, depending on the complexity of the target module’s control logic. Therefore, a more compact notation usually is required to represent test cases. TPL/F uses a built-in macro processor which allows the general form of a test case to be specified as a macro prototype and each specific test case is represented by a single macro call. In practice, test procedures tend to contain only a small number of forms of test cases (often only one) which are invoked many times with different specific data values. Figure 6 shows the three test cases of Figure 5 as they would appear when coded as TPL/F macro calls.

THE TPL/F TEST LANGUAGE

The previous section introduced the control structure and the four major components of TPL/F test procedures (test cases, subprograms, I/O simulators, and macros). This section contains an informal and intuitive introduction to all of the constructs of the TPL/F test language. Complete descriptions of the TPL/F language and the Fortran test procedure system are contained in References 4 and 5.

Figures 7 thru 13 describe the syntax of TPL/F by means of syntax graphs. Rounded boxes represent terminal symbols of the language. Rectangular boxes are non-terminal symbols requiring further definition. The following Fortran syntactic forms (marked with an asterisk in the syntax
Test procedures

Test cases are the primary element of the control structure of a test procedure (Figure 7). Executing a test procedure consists of executing each of its test cases. Subprograms and I/O simulators defined in the test procedure only influence the performance of test cases. Fortran declaration statements may be included in test procedures to define COMMON blocks whose variables are referenced by the test procedure.

The MODULES statement names all of the target modules affected by the test procedure. The first name on the MODULES statement is the primary target module. All test procedure references to target program variables or statement labels, not qualified by an explicit target module name, are directed to the primary target module.

Test cases

Each test case (Figure 8) in a test procedure causes an actual execution of the target program.

The EXECUTE statement specifies where to begin and where to terminate a test case execution, and how many times to execute it. For example, the following statement would cause its test case to be executed three times, each time beginning at statement label 50 of the primary target module and stopping whenever either statement label 25 of target module SUB2 or statement label 35 of target module SUB3 is executed

EXECUTE*3 FROM :50 TO SUB2:25,SUB3:35

An exclamation mark preceding a termination label reference means that the test case is to terminate without executing that statement. Otherwise, it would be terminated immediately afterward. The beginning statement, when specified, is always executed, regardless whether or not its reference is preceded by an exclamation mark. If no beginning label is specified, execution starts at the first executable statement of the primary target module. Regardless whether or not termination labels are specified, test cases always terminate whenever a STOP statement or a RETURN statement in the primary target module is executed.

All test case statements other than EXECUTE, VERIFY, and USE are executed immediately prior to the start of a test case execution. These are usually modified Fortran statements (see below) supplying initial values to the target program.

The VERIFY statement specifies an assertion about the target program. The AT clause, if present, specifies when, during the test case execution, the assertion is to be verified. The following statement causes an assertion to be verified whenever statement label 100 in SUB1 is executed, as well as immediately after the test case terminates.

VERIFY AT SUB1:100,#TERM (SUB1:N .EQ. 0)

An exclamation mark preceding an AT clause label reference causes the assertion to be verified immediately before executing the specified statement. Otherwise, verification occurs after executing the referenced statement. When no AT clause is specified, the assertion is verified after the test case terminates. An attempt to verify a false assertion ter-
minates the current test case (but not the test procedure) and causes an error message to appear in the test execution report as in Figure 3.

The USE statement modifies the mapping of target program I/O references to I/O simulators and affects only the test case in which it appears. For example, if a test procedure contains I/O simulators for logical units 20 and 30, all target program I/O references to these logical unit numbers would ordinarily invoke I/O simulators for those units in the test procedure. The following statements appearing in a particular test case, however, would direct all target program references to logical unit 20 to I/O simulator 30; and all target program references to logical unit 30 to an actual logical unit of that number.

USE 30 FOR 20
USE #ACTUAL FOR 30

The #SCRATCH clause directs target program I/O operations to a temporary scratch file.

Subprograms

Fortran subroutines and functions defined in a test procedure (Figure 9) may be referenced inside the test procedure or may be used as subprogram stubs.

When referenced inside a test procedure, subroutines may be used to perform complicated recurrent test case initialization tasks or Boolean functions may be used to verify complicated assertions about the target program. When used as subprogram stubs, subprograms defined in the test procedure are surrogates for missing subprograms called by the target program. Any subprogram defined in a test procedure can be used in either manner.

I/O simulators

I/O simulators are used to simulate I/O devices and files referenced by the target program. An I/O simulator (Figure 10) is a closed subroutine and is entered each time its logical unit number is referenced by a target program I/O statement.

Each I/O simulator has a unique record counter which always points to the current record on the simulated I/O unit. Prior to each test case execution, all record counters are reset to zero. While executing a test case, each time the target program reads or writes the simulated unit, its record counter is incremented.

I/O simulators can simulate either random or sequential devices. The NO RESET clause on the I/O SIMULATOR statement preserves the record counter value from one test case to another. The NO REWIND clause inhibits target program REWIND statements on the I/O simulator. Examples:

I/O SIMULATOR 12 RANDOM(1000)
I/O SIMULATOR 16 NO RESET

The first statement defines a random I/O simulator for logical unit 12 with a maximum record index of 1000. The second statement defines a sequential I/O simulator for logical unit 16 whose record counter is not reset between test case executions. The latter form is used for providing multiple test cases with different data files from the same logical unit.

The RECORD statement is the primary I/O simulator statement for simulating I/O devices and files. When the target program executes a READ or WRITE statement on a simulated logical unit, the target program is interrupted and control passes to the I/O simulator. The I/O simulator executes until an operable RECORD statement is encoun-
**A Language for Specifying Software Tests**

**Figure 8—Test case syntax**

**Figure 9—Subprogram syntax**

From the collection of the Computer History Museum (www.computerhistory.org)
If the I/O simulator was invoked by a target program READ statement, input data values from the RECORD statement are passed to the target program READ list and control is returned to the target program. If the I/O simulator was invoked by a target program WRITE statement, the data output by the WRITE statement is captured and compared with data in the RECORD statement. If the comparison is satisfied, control returns to the target program; otherwise an error message is written in the test execution report.

RECORD statements that specify a record range are operable only when the value of the host I/O simulator's record counter falls within the specified range. Otherwise, the RECORD statement is treated as a no-op. A record range may be specified as a single record number as in

```
RECORD (5) . . .
```

or as a minimum and maximum value as in

```
RECORD(5:10) . . .
```

The first statement is operable only for the fifth record on the simulated device and the second statement for the fifth thru tenth. A RECORD statement with no record range is always operable.

A RECORD statement may describe either formatted or unformatted records, depending on whether or not a format reference, indicated by a #, is specified, as in the following examples.
Figure 11—TPL/F modifications to embedded Fortran statements

Figure 12—Macro syntax
The first two statements simulate formatted records and the third, an unformatted record. Formatted records must be generated in response to target program formatted I/O statements and unformatted records must be generated in response to target program unformatted I/O statements.

The #ANY clause in a RECORD statement verifies that the I/O simulator was invoked by any target program write operation. The #NOLIST clause verifies that the I/O simulator was invoked by a formatted write operation with no I/O list and is used primarily for skipping titles and formatting information when verifying formatted output records.

The #ERR clause in a RECORD statement simulates an I/O error and causes the ERR=label exit to be taken from the corresponding target program I/O statement. The #END clause simulates an end of file condition and is used to verify a target program ENDFILE statement or cause the END=label exit to be taken from a target program READ statement.

The RETRIEVE statement is used for verifying complex assertions about target program outputs. When the host I/O simulator is invoked by a WRITE or PRINT statement, the output record is captured, optionally interpreted according to a format, and assigned to variables in an I/O list. While the RECORD statement is limited to simple tests of equality between output variables and known values, the RETRIEVE statement makes output records available for any type of test. The first example of the following section (EXAMPLES) illustrates the use of the RETRIEVE statement.

TPL/F modifications to embedded Fortran statements

Fortran statements may appear throughout test procedures. The initialization part of a test case definition, for example, usually consists of Fortran code only. When appearing inside a test procedure, any Fortran statement may be modified as follows.

External identifiers, statement reference counters, and system variables (Figure 11) may be referenced wherever a variable would usually appear in a Fortran statement.

An external identifier refers to a variable or array in the target program. The initialization code and assertions in the example of the previous section contained external identifiers pointing to the variables N1 and N2 of target module IOSUB.

Statement reference counters provide a means for verifying the number of times a target program statement is executed. Every test procedure reference to a statement reference counter causes an integer-valued counter to be associated with the indicated statement in the target program. Prior to each test case execution, all statement reference counters are reset to zero. During test case executions, statement reference counters are incremented each time their associated statements are executed. The following
statement, for example, verifies that the statement at label 100 of target module SUB1 was executed exactly five times during the current test case execution.

```
VERIFY (#SUB1:100 .EQ. 5)
```

Three special symbols are used to reference system variables whose values are controlled automatically by the test procedure processor.

The value of `#TEST` is the index (e.g., first, second, etc.) of the currently executing test case.

`#REF` is defined as follows. Every subprogram and I/O simulator has a reference counter associated with it. All reference counters are reset to zero prior to each test case execution and incremented by one on each entry to their associated subprograms. Inside a subprogram or I/O simulator, `#REF` has the value of the subprogram or I/O simulator's reference counter. In all other places, `#REF` is always equal to zero.

Inside an I/O simulator routine, `#RECORD` has the value of the I/O simulator's record counter. Elsewhere, the value of `#RECORD` is always zero. Examples:

```
Y=VAL(#REF,#TEST)
IF (#RECORD .EQ. N) GO TO 100
```

Ordinarily, Fortran I/O statements embedded in a test procedure are not affected by I/O simulators. A logical unit designator prefixed by a pound sign in an embedded Fortran I/O statement temporarily suspends this rule and causes the I/O statement to be treated the same as a target program reference to the logical unit. For example, the following statement appearing inside a test procedure references the I/O simulator for logical unit 10, not the actual logical unit 10.

```
READ (#10) X,Y,Z
```

The ABORT statement is used inside subprograms and I/O simulators to terminate the current test case when an error is detected. Terminal assertions of aborted test cases are not verified.

**Macros**

The primary role of macro definitions (Figure 12) is that of an efficient shorthand notation for describing test cases. Macro definitions are entered in a macro definition library under the symbolic name specified on the macro statement. When invoked by a reference to the macro name, formal arguments appearing in the text of the macro definition are replaced by actual arguments supplied by the call. The modified text of the macro definition then replaces the macro call statement.

The conditional construct (Figure 13) allows statements to be conditionally included in a macro expansion. The Boolean expression in the COND statement is evaluated each time its host macro is expanded. If the expression evaluates to true, all statements following the COND statement, down to the ELSE statement, if present, are included in the macro expansion. When the COND statement expression evaluates to false, only those statements between the ELSE statement, if included, and the ENDC statement are included in the macro expansion.

String terms in COND statement expressions are either quoted strings or formal arguments of the host macro definition which are replaced at macro expansion time by strings.

Two special forms of numeric terms may appear in COND statement expressions. `#NARG` is the number of actual arguments supplied in the current call on the host macro. The numeric value of `#VAL(arg)` is determined as follows. Usually, arg is a formal argument in the host macro definition. After argument substitution, if arg is a numeric constant, then that is the numeric value of `#VAL(arg)`; otherwise the macro expansion terminates in error. Figure 14 illustrates conditional macro expansions.

**EXAMPLES**

The test procedure example in the beginning of this paper illustrates the basic structure of test case definitions, the use of macro definitions for replicating test cases, I/O simulators and subprogram stubs. Two additional examples in this section illustrate common styles used for writing test procedures.

Figures 15 and 16 illustrate the use of I/O simulators for testing I/O-oriented target modules. The target module of

```
1 READ (12,100,END=300) X,Y
200  
300 WRITE (6,150) W
STOP
END
```

**Figure 15—An I/O-oriented target module**
MODULES MAIN
C
C INPUT VALUES
C
I/O SIMULATOR 12 NO REWIND
RECORD (1) #1, 0.0,0.0
RECORD (2) # END
RECORD (3) #1, 0.01,0.00001
RECORD (4) #1, 0.01,0.00002
RECORD (5) # END
RECORD (6) #1, 1.0,5.0
RECORD (7) #, 1, 2.0,4.0
RECORD (8) #1, 3.0,3.0
RECORD (9) # END
RECORD (10) #1,10000.0,0.00001
RECORD (11)# END
1 FORMAT(, BASE', FI2.6, I0X, 'VALUE', FI2.6)
END
C
C OUTPUT VALUES
C
I/O SIMULATOR 6 NO REWIND
DIMENSION X(4)
DATA X/10.0, .0.0132, 2.5, 1.01/
RETRIEVE (#1)
IF (ABS(VAL-X(#TEST» .GT. 0.00005) ABORT
RETURN
1 FORMAT('VALUE =', FI4.8)
END
C
C DEFINE GENERAL TEST FOR MAIN
C
MACRO TSTPRG(ZP)
VERIFY AT :200 (:ABS(Z-ZP) .LT. 0.0005)
EXECUTE
MEND
C
FOUR TEST CASES
C
TSTPRG(1.0)
TSTPRG(1.001)
TSTPRG(5.0)
TSTPRG(100.0)
FIN
Figure 16—A test procedure for the target module of Figure 15

Figure 15 is a main program that reads formatted records from logical unit 12 until an end-of-file condition is encountered. After some processing, the program writes one formatted record on logical unit 6 and terminates.

The test procedure in Figure 16 exercises the single target module, MAIN. Four test cases each supply inputs to MAIN on logical unit 12 and verify its outputs on logical unit 6. Additionally, a non-terminal assertion is verified each time control reaches MAIN:200.

Figures 17 and 18 contain another example illustrating the use of internal references to I/O simulators for generating large amounts of data for test case initialization and verification. The target module, SUB2, of Figure 17 performs a transformation on the array M.

The test procedure of Figure 18 contains four test cases, each of which supplies SUB2 with 24 input values for the array M and verifies the contents of M returned by SUB2. The input and output values are contained in two I/O simulators read only by the test procedure.

DISCUSSION

The notation that software tests can be specified in a formal language, just as are computer programs, will likely improve the quality of software test design since, for the first time, software tests reside in a medium (the test pro-

SUBROUTINE SUB2(M)
DIMENSION M(8,3)
FIN
Figure 17—A target module

MODULES SUB2
C
C INPUT VALUES
C
I/O SIMULATOR 10 NO REWIND
RECORD (1:3) 0,0,0,0,0,0,0,0
RECORD (4:6) 1,1,1,1,1,1,1,1
RECORD (7:9) 1,0,0,0,0,0,0,0
RECORD (10:12) 0,1,1,1,1,1,1,1,1,1,1,1
END
C
C OUTPUT VALUES
C
I/O SIMULATOR 20 NO REWIND
RECORD (1) 1,0,4,8,0,1,5,0
RECORD (2) 1,1,0,1,5,3,6,0
RECORD (3) 2,0,1,1,8,1,6,4
RECORD (4) 7,9,1,6,4,6,9
RECORD (5) 1,7,9,1,4,1,9,4
RECORD (6) 6,2,5,9,0,3,6,2
RECORD (7) 0,7,2,0,9,6,9,9
RECORD (8) 9,5,7,0,9,1,2,9
RECORD (9) 1,9,0,7,0,0,2,2
RECORD (10) 3,6,8,4,6,5,7,3
RECORD (11) 2,5,5,1,5,8,5,3
RECORD (12) 9,3,3,0,9,9,5,8
END
C
C VERIFY OUTPUTS
C
FUNCTION OUTVFY
WRITE (#20, ((M(I,J), J=I,8), I=I,3))
RETURN
END
C
C DEFINE GENERAL TEST FOR SUB2
C
MACRO TSTSB2
READ (#10, ((M(I,J), J=I,8), I=I,3))
VERIFY (OUTVFY)
EXECUTE
MEND
C
FOUR TEST CASES
C
TSTSB2*4
FIN
Figure 18—A test procedure for the target module of Figure 17
procedure) that is both executable and readable and therefore available for peer review and criticism. Automatic software test drivers should also influence the quality of software test design by providing feedback on the degree of testing thoroughness actually achieved by a test procedure.

Perhaps the strongest effect of automatic software test drivers and formal test procedures will be on the maintenance of production software. It is widely agreed that the majority of errors appearing in production software are introduced as unintended side effects of post-release program modifications. This is due, in large part, to the absence of a convenient mechanism for preserving test cases throughout a program’s life-cycle so that they can be used to check out post-release program modifications. In order to effectively retain software tests for post-release regression testing, it is necessary that someone other than the original test designer be able to execute them and verify the correctness of test results. Formal test procedures should facilitate the retention of software test cases throughout the entire life cycle of computer programs since they are completely self-contained, they execute automatically without special test set-up instructions, and they are self-checking. The adoption of automatic software test drivers should therefore lead to significant software cost savings since it is often estimated that up to 70 percent of the life-cycle costs of computer software are attributable to testing and debugging.

REFERENCES
