A TESTING TOOL FOR A FIRE-CONTROL ENVIRONMENT

Victor R. Basili
Robert E. Noonan
Department of Computer Science
University of Maryland
College Park, Md. 20742

This paper describes the design of a testing tool to be used by an independent validation/verification group for fire-control software. The basic approach is to generate for each test case an execution histogram of the program; this must be done without modifying the object program. Using computer generated reports showing the completeness of testing so far achieved, the tester is aided in constructing a minimal effective test set.

Introduction

The most practical tool in the validation and verification (V/V) of medium to large scale software systems is to test the system using a carefully chosen set of test data. The test sets can be chosen for two purposes: to show that the program satisfies its specifications, and to show that the program operates correctly. At the present time, little work has been published about the generation of test cases to validate the specifications.

In recent years a great deal of research has been devoted to studying the problem of generating test data to show that a program operates correctly [1,2,3]. It is well known that if the tester views the program as a black box, then complete testing of the program is impossible in practice. For example, if one considers the problem of checking a program which multiplies two 36-bit integer numbers, the time required on a modern high speed computer is on the order of billions of years. However, not all test cases are of equal value. Taking the structure of the program into account, one can choose a reasonably small set of test cases which provide a high level of confidence in the reliability of the program.

One criteria for choosing test cases is to insure that every statement has been executed at least once and that every loop is executed 0,1 and n times (if possible) where n is some relative maximum value. This provides the tester with the knowledge that every line of code is reachable, was executed at least once to give correct results, and can lead to termination.

One view of a testing system is as a man-machine interaction. For example, some aspects of this task are better handled by the machine:

(1) executing the test cases,
(2) recording which test cases have been run,
(3) recording for each test case which statements were executed,
(4) compiling over all test cases the execution history of the program,
(5) informing the tester which statements have never been executed.

Relative to the state of the art, other aspects might best be handled by man. These include:

(1) test case generation
(2) selection from the test cases of a minimal effective set.

The above criteria for program testing were used in the design of a test tool for a fire-control environment.

Project Background

The Navy, in developing a new submarine-based intercontinental ballistic missile system called the Trident, is writing new fire-control software. The Naval Surface Weapons Center/Dahlgren Laboratory is responsible for the development of this software. Organizationally, there are separate groups with responsibility for the development and for the validation/verification (V/V) of the software. The independence of these two groups is an important organizational constraint within which a testing tool was to be developed.

The goal of the V/V group is to test the programs in an environment as close as possible to the actual environment. Note that this goal can not be realized absolutely since to do so would necessitate actually firing the missiles. Other than the fact that the missile firing hardware is simulated by a computer, the actual test environment must be identical to the one on board the ship.

In meeting this goal, the V/V group operates under fairly rigid constraints.

(1) For confidence in the correctness of the production system, the object program tested must be identical to the production program. Additional code cannot be inserted or existing code modified for testing purposes.
(2) The nature of the programs and of the onboard (fire-control) computer system is such that the assumption has to be made that little or no memory space is available for testing requirements.

These two constraints help to insure that not only the programs but also their environment are substantially the same under testing as they are under actual operational conditions. These constraints made the development of a testing tool using the onboard computer very difficult.

Fortunately, an emulator for the onboard computer was available on a Nanodata QM-1. The latter machine had more memory than the onboard computer and any software required for testing purposes could be reasonably constructed using a combination of microcode and machine code. Thus, it was decided to develop a testing tool using the QM-1.

The programming language being used is an ALGOL-like expression langauge, called THLL[4], which was developed expressly for this project. The compiler for THLL runs on a large, batch machine, the CDC 6700 and produces assembly code output. Since the compiler was developed using a translator writing system, it was theoretically possible to instrument the compiler to generate additional code for testing purposes. However, given the constraints cited above, this was not a viable option, since it would have meant modifying the programs considerably and expanding their size.

Throughout the remainder of this paper, the term compilation system will be used to refer collectively to all the passes required to produce an absolute deck, that is, the passes of compilation, assembly, linkage editing, etc. In addition to the absolute deck, the compilation system also produces an extended attribute/cross-reference table called the Schema. This table contains the absolute machine addresses of each variable, label, and procedure, as well as the first instruction of each high level language statement. The existence of this table is essential given the constraints already cited.

Overview

The purpose of this section is to describe the Trident Testing Environment job flow (see Figure 1). The programs to be tested are written in the high level language THLL. They are run through the compilation system on the CDC 6700 producing as output the Schema data and the object program for the onboard computer.

The instrumentation program modifies the object program to allow an execution histogram to be built as the program is run. This is accomplished by replacing key instructions in the flow of control of the object program by trap instructions, and storing any related information, such as the instruction replaced, in the Breakpoint table. The key instructions chosen and their locations are determined from the Schema data. This process is discussed in detail in the next section.

The instrumented program is then run on the QM-1 with a variety of test data producing results and a Test Run Table which keeps track of the parts of the program exercised by the particular test data. The data produced by each test run is accumulated in a library system called the Scoreboard. It allows a systematic record to be kept of the completeness of testing of a program with respect to the criteria that

(1) each statement is executed at least once
(2) each loop is executed 0,1,n times (if possible).

Later sections contain a discussion of how data is collected for the Test Run Table and describe the report generated by the Scoreboard.

Once the Scoreboard has collected a sufficient set of test cases that exercise the program completely to the testor's satisfaction on the QM-1, the set is refined to a minimal but complete subset. This allows the testor to minimize the time needed to test the programs on the onboard computer. Since the amount of time for final validation of the programs on board the actual ship is severely limited, the use of the QM-1 to develop and "debug" a good set of test cases was of added benefit. Results from running the programs on the QM-1 are gathered from the Scoreboard and then compared against results obtained from running the programs on the onboard computer. This permits final testing in an environment almost identical to actual operating conditions.

For illustration, a specific example will be used throughout the sections which follow. The example chosen is the well known algorithm of Euclid for calculating the greatest common divisor of two numbers. The THLL program embodying this algorithm is given in Figure 2. The assembly/object program generated by the THLL compiler is given in the Appendix.

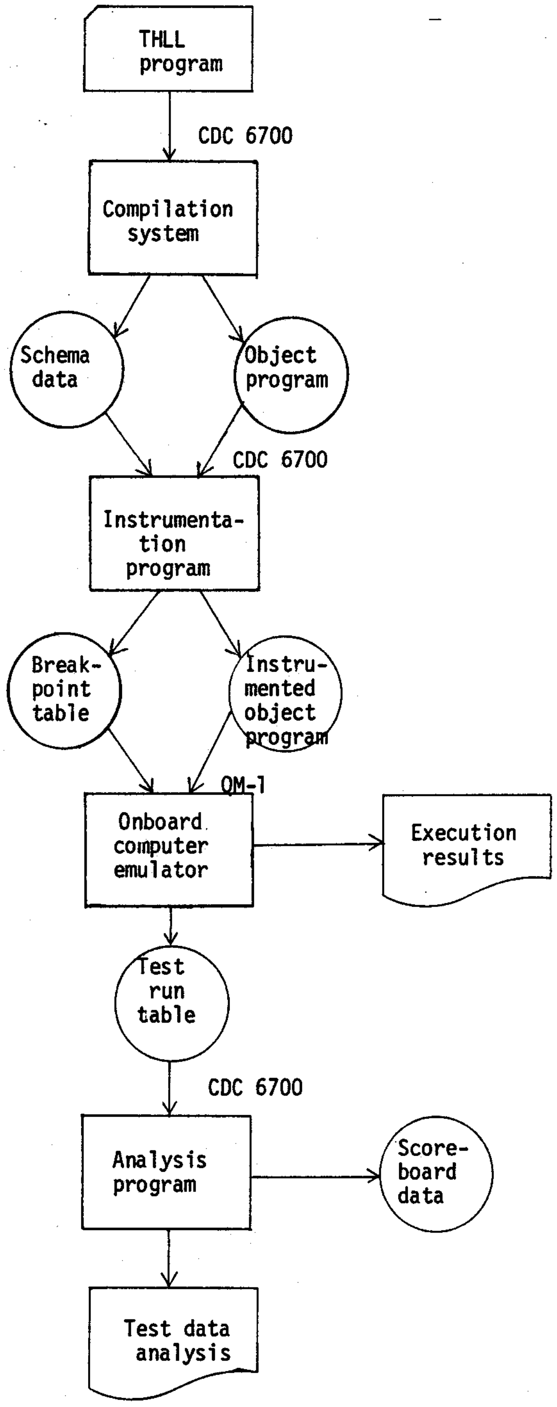

Figure 1. Trident Job Flow for V/V

Instrumenting the Object Program

The execution history of the program is kept as a basic block historgram. A basic block is defined to be a sequential set of statements with a single entry and single exit point and which contains no basic blocks. Thus each statement in a basic block is executed an equal number of times. In THLL potential basic blocks (as defined by their first statements) are:

```
Line Program
 20  INTEGER PROCEDURE EUCLID (A,B);
 21  VALUE A,B;
 22  INTEGER A,B;
 23     /* THE VALUE IS THE GREATEST COMMON
 24        DIVISOR OF A AND B */
 25  BEGIN
 26  INTEGER R,C;
 27  A = ABS (A);
 28  B = ABS (B);
 29  IF A < B THEN BEGIN C = A; A = B; B = C END IFEND;
 30  WHILE B NEQ 0 DO
 31     BEGIN
 32     R = A MOD B;
 33     A = B;
 34     B = R;
 35     END;
 36  RETURN A;
 37  END; /* END OF PROCEDURE EUCLID */
```

Figure 2. THLL Euclid Program

procedure entry
any labelled statement
the then and else parts of an if statement
the body of a while or for loop
the subcases of a case statement
the first statement after any multibranch compound statement, i.e., an if, for, while, or case statement.

The use of a basic block histogram rather than a statement histogram saves both time and space with no loss of information.

The basic blocks (and their associated source line numbers) for the example program (Figure 2) are given in Figure 3. The program contains only 5 basic blocks, reflecting its simple structure. Note that the type of basic block is solely determined by the first statement of the block.

The instrumentation program is responsible for calculating the basic blocks of the program. The process consists of identifying relevant branch points generated by the THLL compiler. This data is readily available in the Schema as a set of label points, classified as actual, pseudo, or phony labels. An actual label is one that exists in the source code and is mapped directly into assembly code; procedure names and statement labels are examples of actual labels. A pseudo label is one that is generated by the compiler for internal branching; examples include the else part of an if, the endif join, etc. It should be noted that not all pseudo labels generated correspond to basic blocks; for example, the pseudo label for the while test is not used since its execution count can be inferred from that of the body of the loop itself. A phony label is one that is required to identify certain types of basic blocks but no label is generated by the compiler because no branch is needed; examples include the then part of an if, the first subcase of a case, and the body of a while loop. This information is summarized in Figure 4.

Basic block	Lines	Type of basic block	Remark
1	20-29	proc entry	proc EUCLID
2	29	then part	if on line 29
3	29-30	join/endif	end of if on line 29
4	30-35	loop body	while loop on line 30
5	35-37	join/endloop	end of while loop on line 30

Figure 3. Basic Blocks of Example

Type of basic block	Location associated with basic block is the first instruction:	Type of label
proc entry	of entry macro	actual
labelled stmt	of labelled stmt	actual
then part	of then part	phony
else part	of else part	pseudo
join/endif	after if stmt	pseudo
loopbody/for	of loop body	pseudo
loopbody/while	of loop body	phony
join/endloop	after loop	pseudo
subcase/first	of first subcase	phony
subcase/other	of all but first subcase	pseudo
join/endcase	after case stmt	pseudo

Figure 4. Basic Block Information

Figure 5 gives the relevant Schema entries both for the actual basic blocks (given in Figure 3) and for the potential basic blocks. An example of the later is the potential block corresponding to the else part of the if statement on line 29 of Figure 2; however, as can be seen in Figure 2, the if statement has no else part. However, the THLL compiler generates a pseudo label for the else part (L14$$) anyway. In order to save space and avoid confusion, all such empty basic blocks are eliminated (via a simple analysis).

Index	THLL line	Type	Assembly line	Assembly location	Assembly label
1	20	proc entry	21	41E0	EUCLID
2	29	then part	39	41F4	-
3	29	else part	43	41F8	L14$$
4	29	join/endif	44	41F8	L12$$
5	30	loopbody	50	41FB	-
6	35	join/endloop	63	4203	L18$$

Figure 5. Relevant Schema Labels

The instrumentation program records the relevant information about each basic block in the Breakpoint table. The index into the table is the block number. Each entry contains the starting address, called the <u>breakpoint location</u>, of each basic block; this location is used to modify the object program. The <u>loopend</u> entry gives the block number of the loop body if the entry is a join/endloop; otherwise it is zero (the need for this entry will be explained in the next section).

The object program is instrumented by storing a trap instruction at each breakpoint location. The instruction replaced is saved in the appropriate entry of the Breakpoint table. The trap instruction inserted contains the basic block number for easy reference into the Breakpoint table at run-time. The Breakpoint table for the example program is presented in Figure 6.

Basic block	THLL line	Breakpoint location	Type of basic block	Loopend	Saved instruction
1	20	41E0	proc entry	0	32370000
2	29	41F4	then part	0	E2F70013
3	29	41F8	join/endif	0	36FF0000
4	30	41FB	loopbody	0	18E70010
5	35	4203	join/endloop	4	18F70010

Figure 6. Breakpoint Table

Testing the Program

Once the object program has been instrumented and the Breakpoint table has been built, the program is capable of being executed under test conditions. As each breakpoint is trapped by the hardware, the trap handling library routine is executed. This is defined as follows:

```
/* Find the basic block number (BBN) of the break-
     point which caused the trap.  It is stored in
     the trap instruction. */
   assign (BBN)
/* Increment the COUNT for the basic block entered */
   COUNT (BBN) := COUNT (BBN) + 1
/* Perform only for entries which are loop ends for
        loops */
   if LOOPEND (BBN) ≠ 'no'
      then /* let LOOP designate the BBN of the loop
                itself */
         LOOP := LOOPEND (BBN)
         /* save the maximum of the number of times
                the loop was executed up to that
                point */
         MAXCOUNT (BBN) := maximum (MAXCOUNT (BBN),
                COUNT (LOOP))
         /* record if loop executed 0 times last
                time */
            if COUNT (LOOP) = 0 then EXECO (BBN) :=
                  1 fi
         /* record if loop executed 1 time last
                time */
            if COUNT (LOOP) = 1 then EXEC1 (BBN) :=
                  1 fi
         /* reinitialize count for next run */
            COUNT (LOOP) := 0
      fi
 /* execute the instruction replaced by the trap */
    execute (SAVEDINSTRUCTION (BBN))
 /* return control to the instruction after the
      breakpoint */
    return
```

The Test Run table is a file which is created to record the history of the test run. It contains all the histogram information generated by the execution of the program along with identification information. The latter includes specification of the specific program, date, a unique number assocaited with the test run and a copy of the test data. The histogram information includes the block number, the loopend field and, the count of the number of times the block was executed. If the block is a loopend it will also include information on whether the loop was executed 0 (Exec0) or 1 (Exec1) times and the maximum number of times the loop was executed (Maxcount) for any exit from the loop. As an example consider the testing of the procedure EUCLID. A sample test run table for a pair of values for the formal parameters A and B is given in Figure 7.

Program:	EUCLID	Test Run:	1
Version:	1	Test Data:	129, 1 (A,B)
Date:	02/10/76	Results:	1

Basic block	Source line	Type of basic block	Count	Loopend	Exec 0	Exec 1	Max-count
1	20	proc entry	1	0			
2	29	then part	0	0			
3	29	join/endif	1	0			
4	30	loop body	1	0			
5	35	join/endloop	1	4	0	1	1

Figure 7. Test Run Table

Interrupting the program at each breakpoint can be very time consuming, especially if the breakpoints are contained deep within a nested set of loops. Therefore it would be worthwhile to be able to turn off breakpoints dynamically when a sufficient amount of information about the testing of certain parts of a program has been accumulated. This can be done in several ways. One way is to alter the above routine to turn off breakpoints once the count has exceeded a certain number. For example after the count has been incremented, a check can be made of the form:

```
if COUNT (BBN) > NUMBER
   then /* store the SAVED INSTRUCTION in the
           breakpoint location.  Assume LOC is the
           location of the instruction which caused
           the trap. */
           MEMORY (LOC) := SAVED INSTRUCTION (BBN)
fi
```

This would eliminated the trap and the program would no longer interrupt at that location.

It is also possible to set traps only at a particular level, say the procedure level. Thus time and space permitting some trapping could be done for testing programs on the onboard computer and the intermediate results obtained could be compared with the intermediate results from the QM-1 runs. This would provide a good consistency check at levels lower than the entire program.

Analyzing the Test Results

After the program has been executed on some test data on the QM-1, the Test Run table produced is transported back to the CDC 6700 for analysis and report generation. The process is a simple one:

(1) Update the cumulative data for this program in the Scoreboard.
(2) Add the results of the test run to the Scoreboard.
(3) Output the Test Analysis report.

From a logical viewpoint the Scoreboard can best be viewed as a single hierarchical file organized by program id (which must be unique). There are two basic levels to this file: program level and test set level. Information at the program level includes the program id, version number, number of test cases, and the cumulative basic block histogram. The test set level is organized by test run number and includes the test data, the results, and the basic block histogram for the test run.

Figure 8 shows a sample of the basic report after two test runs. The data for the first test run was (129,1) and for the second (129,5). The Test Run Table for first test set was given in Figure 7, while the table is implicit for the second test set in the report in Figure 8. The cumulative figures combine these two test cases. Note that in the final report the loop information is printed with the loop body although it was originally stored with the loopend. The column labelled COUNT shows the count of the number of times that the block was executed for this run and the total number of times executed cumulatively.

Note that the report shows that the test sets so far have never executed the then part of the if on line 29 and the loop has always been executed at least once. Thus, additional test cases are necessary.

Program: EUCLID Test Case: 2
Test Data: 129,5 Date: 02/10/76
Results: 1

(THIS RUN/CUMULATIVE)

Block				LOOP ********************		
basic	Line	Type	Count	Maximum	Once	0 Times
1	20	proc entry	1/2			
2	29	then part	0/0			
3	29	join/endif	1/2			
4	30	loop/body	3/4	3/3	N/Y	N/N
5	35	join/endloop	1/2			

Figure 8. Test Data Analysis

Conclusions

The system proposed in the paper meets the goals of providing the tester with an aid for determining the effectiveness of particular test cases relative to a particular set of test criteria. It provides the tester with the information he needs in order to decide whether additional test cases are needed. Because of the nature of a fire-control environment, it allows the tester at each stage of validation to choose a minimal effective subset of test cases to run.

The system is an example of tailoring standard techniques to fit a specific environment. Although a simpler approach to collecting an execution histogram is to have the compiler insert the necessary code, this was not a viable option due to the fire-control constraints. We had to find some other method of achieving the same effect. The approach chosen of introducing breakpoints into the object code was independent of the test criteria chosen.

We have tried to give some of the algorithms required to build this system. Algorithms which were not given include:

(1) The algorithm for deciding when to automatically turn off certain breakpoints.
(2) Methods of identifying the results of a procedure or set of procedures. Clearly, results include any value output or any value returned from a procedure.
(3) Methods of comparing results from different machines. In a fire-control environment, algorithms are normally developed on one machine, and recoded in a different language for a different machine.

Acknowledgments

This work was supported by the Naval Surface Weapons Center/Dahlgren Laboratory.

References

1. Hetzel, W. C. Program Test Methods. Prentice-Hall (1973).

2. Huang, J. C. An approach to program testing. Computing Surveys, 7 (September 1975), pp. 113-128.

3. Goodenough, J. B. and Gerhart, S. L. Toward a theory of test data selection. IEEE Transactions on Software Engineering, SE-1 (June 1975), pp. 156-173.

4. Ormstron, A. Trident High Level Language User's Guide, NSWC/DL TN-DK-37/75 (December 1975).

Appendix: Assembly Listing of Program EUCLID

Line	Location	Label	Opcode	Operands
21	41EC	EUCLID	RES	0
22	41E0		ENTER	EUCLID,2
23	41E8		L,9	$0,1
24	41E9		S,9	16,7
25	41EA		L,9	$1,1
26	41EB		S,9	17,7
27	41EC		S,2	3,7
28	41ED		SHLS,6,7	-1,0
29		*	LINE	25
30		*	LINE	27
31	41EE		LM,15	16,7
32	41EF		S,15	16,7
33		*	LINE	28
34	41F0		LM,14	17,7
35	41F1		S,14	17,7
36		*	LINE	29
37	41F2		SBR,13,15	14
38	41F3		BCGE	L14$$
39	41F4		S,15	19,7
40	41F5		S,14	16,7
41	41F6		S,15	17,7
42	41F7		B	L12$$
43	41F8	L14$$	RES	0
44	41F8	L12$$	RES	0
45		*	LINE	30
46	41F8	L20$$	RES	0
47	41F8		LIRA,15	0
48	41F9		CA,15	17,7
49	41FA		BCE	L18$$
50		*	LINE	31
51		*	LINE	32
52	41FB		L,14	16,7
53	41FC		SHADN,13,14	32
54	41FD		DV,13	17,7
55	41FE		S,14	18,7
56		*	LINE	33
57	41FF		L,13	17,7
58	4200		S,13	16,7
59		*	LINE	34
60	4201		S,14	17,7
61		*	LINE	35
62	4202		B	L20$$
63	4203	L18$$	RES	0
64		*	LINE	36
65	4203		L,15	16,7
66	4204		RETURN	2
67		*	LINE	37
68	4205		LIRA,15	0
69	4206		RETURN	2
70	4207		EXIT	EUGLID,2,20