MICROTEST — A Testing Tool on PC*

István Forgács, Attila Horváth and Endre Somos

Computer and Automation Institute
Hungarian Academy of Sciences
H—1518 Budapest, Pob. 63 Kende u. 13—17, Hungary

Abstract

The MICROTEST system is described, which is a tool for decentralized "distributed" testing of application programs targeted to run on large mainframe computers. It detaches the testing process from the host environment, thus the programs can be tested on PC. The system includes a high-level test language for validating program results against the specification. Unlike the other testing tools, it does not use instrumentation but it compiles the source program into an object code which is interpreted. (This is the key to the transportation of testing to PC.) MICROTEST has five components. The Static Analyzer compiles the source program into an intermediate language, and produces data for a series of quality metrics. The Static Report Generator makes the Static Usage, the Branch and the Module Quality reports. The Assertion Compiler compiles the test program into the same intermediate language as the one used by the Static Analyzer. The Dynamic analyzer links the compiled programs and interprets it. The Dynamic Report Generator makes the Test Log, the Dynamic Test Path, the Data Coverage, and the Program Coverage reports.

Introduction

In the last few years the theory and practice of program testing came into prominence. The reason of the research is in the recognition of the fact that maintenance cost is the strongest component of software development expencies. Since error correcting is about two third of the entire cost of maintainability, the great effort seems intelligible [1, 2, 3].

There is a lot of testing techniques in the theory, but in the practice only three of them are used:

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

— structural testing,
— functional testing,
— code reading.

In case of structural testing, the program is investigated as a white box. The test cases are created based on the flow-graph of the program to reach a high program coverage ratio. Many criteria for program coverage were suggested from statement testing to path testing. The partial ordering of these strategies are in [4, 5]. In [4] a worst case estimation is given for the number of test cases.

In case of functional testing the program is investigated as a black box. The test cases are gathered from the specification and from other user information named 'oracle' [6, 7].

In case of code reading the function of the program is determined manually and compared against the specification. Although this method seems the less effective, experience shows that code reading sometimes can do better then the two others [8].

The development of testing tools

Most of the tools are based upon instrumentation. During instrumentation, traps or counters are inserted into the source code to measure the program coverage. These tools work on the basis of the structural testing method.

In case of earlier tools the test data was typed manually from the keyboard, while the result was read from the screen. The inserted counters measured the test coverage.

The next step was the automatic validation of the program. The validation procedures (assertions) were built into the program conditionally, thus the program could be executed with or without validation. (E.g. in [9], assertions were written as special comments, and an optional preprocessor compiled them into the source code.) This way the output can be validated but the input is not a basic component of the validation system (especially in the case of keyboard input). These assertion instructions were the predecessors of the test languages.

Since one of the essential requirements of testing is the reproducibility of tests, it is necessary to store the test data for all the test cases. This is done in some testing tools (e.g. in [10] the inpu tis received from the keyboard, it is recorded in a file, and when the test is repeated, this file substitutes the input device).

The last generation of testing tools detaches the description of test data (both input and output) from the program. It provides an opportunity to generate the test data from the specification itself (either manually or automatically) even before the implementation of the program. This method summarizes the advantages of structural and functional testing.

An example is the assertion language of the SOFTEST system [11] which uses first order predicate logic to define the program specification. The program behavior is specified in terms of PRE and POST assertions for the data states and INPUT/OUTPUT assertions for the data base file accesses. The test data is generated from the PRE and INPUT assertions and the test results are checked against the POST and OUTPUT assertions. All the assertions can be given in the forms of individual values, sets, ranges, functions, and relations. The SOFTEST system instruments the object code with assembly routines to execute the assertions. Directly the program variables are used, therefore the object code of the program should be handled by the testing system, which decreases the portability.

Now we can summarize the requirements of a good test system:

— the input and output test data should be stored separately from the program under test,

— the test data should be described with the help of a high level test language which supports all the conditional and cyclic data assignments,

— both the program under test and the test program should be compiled and linked into one executable or interpretable code,

— the system should be independent from the hardware environment,

— both the executed program and all kinds of data should be under complete control of the test system.

The MICROTEST system

The MICROTEST is a module test system which can test and debug COBOL programs on PC-s. The system contains two compilers to translate both the COBOL and the assert program into the same binary code named AL (assembly like) language. In this AL language the commands are coded binary and the Dynamic Analyzer interprets the commands. For example the COMPUTE $p3=p1+p2$ statement is translated to ADD $p1, p2, p3$ which are four integers in the command table. The first integer stores the code of the addition, the others store the pointers to the $p1$, $p2$ and $p3$.

The system consists of five elements:

— Static Analyzer,
— Static Report Generator,
— Assertion Compiler,
— Dynamic Analyzer,
— Dynamic Report Generator.

The Static Analyzer has two functions. First, it compiles the source code into the AL; second, evaluates the source program to produce data for a series of analytical reports and a number of quality metrics. The Static Analyzer differs from the others not only in compiling the program but in the instrumentation, too. It is needless to instrument the source code. During the static analysis, each I/O and external function calls are translated into an assertion call which calls the appropriate assertion program module. The Static Analyzer was made with the PROFLP compiler generator which was very useful in making the compiler part of the static analyzer for different COBOL versions.

The Static Report Generator produces the following reports:

— Module Quality Report (see TABLE I),
— Static Data Usage Report (see TABLE II),
— Branch Report (see TABLE III).

TABLE I

MODULE QUALITY REPORT

Module name: CMERGE	Date: 12. 5. 1988	Page: 1;

Static Metrics:		
	Data Complexity	= 0.31
	Control Flow Complexity	= 0.92
	Data Flow Complexity	= 0.33
	Interface Complexity	= 0.27
	Portability	= 0.78
	Maintainability	= 0.46
	Testability	= 0.31

TABLE II

STATIC DATA USAGE REPORT

Module name: CMERGE	Date: 12. 5. 1988	Page: 1

Data no.	Lv no	Data Name	Strg Type	Data Type	Data Lng.	Dim	Picture	Data usage					
			file										
1	1	REC—1		disp	10				a		t		
2	2	KEY—1		disp	3		X(3)	p		r			
3	2	SAL—1		disp	7		9(5)V99		a				
4	1	REC—2		disp	10				a		t		
5	2	KEY—2		disp	3		X(3)	p		r			
6	2	SAL—2		disp	7		9(5)V99		a				
7	1	OUTREC		disp	10					r	t		
8	2	KEY—3		disp	3		X(3)						*
9	2	SAL		disp	7		9(5)V99						*
			work										
10	1	W		disp	17								*
11	3	SUM—SAL		disp	10		9(8)V99		a	r			
12	3	MAX—SAL		disp	7		9(5)V99	p	a	r			
13	77	N		comp3	3		9(5)		a	r		i	
14	77	PAR—SAL		disp	7		9(5)V99	p	a	r			
15	77	DISP—RESULT		disp	8		ZZZZ9.99		a	r			
			link										
16	1	LINK—DATA		disp	8								*
17	2	PAR		disp	1		A	p	a				
18	2	RESULT		disp	7		9(5)V99		a	r			

Number of predicates:	5	Number of arguments:	11
Number of results:	9	Number of transients:	3
Number of inits:	1		
	Total number of data items used:	14	
	Total number of data items not used:	4	

The Module Quality Report computes the following static metrics:
— data complexity,
— control flow complexity,
— data flow complexity,
— interface complexity,
— maintainability,
— testability.

The Static Data Usage Report contains information on the data fields which can be gained from the source code: data names, level number, data length, data type, dimension, picture, and data usage. A data can be predicate, argument, result, transient (parameter of an I/O or CALL statement), or it can get an initial value. A '*' character indicates when the data is not used at all.

The Branch Report describes the control flow of the program under test. The branches are identified by a branch number. The report contains the line number, the branch number and the source program lines.

The Assertion Compiler compiles the assertions into the AL language. The original SOFTEST language was developed to a high level language. In our system the assertion variable declaration, FOR, REPEAT—UNTIL, WHILE—DO, and CASE statements permit the sophisticated input and output test data assignment/validation. An example:

```
FOR $i = 1 TO 10 DO
    ASSERT IN abc[$i] E SET(1, 2, $i)
END;
```

At the first call the ten array elements of *abc* get the value 1, at the second call get 2, while at the third call *abc*[1] = 1, ..., *abc*[10] = 10.

The compiler has a built-in editor too.

The Dynamic Analyzer first links the compiled COBOL and test programs then interprets the linked code. This way the Dynamic Analyzer contains a driver, which interprets the object code and produces all the necessary statistics. During the run it reads the test data either from the assert file or from the keyboard, and validates the results against the data read from the assertion file. It produces statistics in order to obtain some dynamic metrics which qualify the program and the testing process.

The MICROTEST Dynamic Analyzer operates in an interactive dialogue mode; the user has the opportunity

— to stop the run at any moment,
— to take breakpoints into the program,
— to use step-by-step execution,
— to display/change any data item.

Though these are debugger functions, they can help the testing, too.

Whenever a validation error occurs (the actual data differs from the prescribed one), an assertion violation interrupt is executed. The error message contains both the COBOL and the assertion line number, the data name as well as the prescribed value or interval, and the actual value.

TABLE III

STATIC BRANCH REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 2

Line no.	Brch. no.	PROCEDURE DIVISION statement
93A	26	∗∗∗ EMPTY BRANCH for statement in line 92 ∗∗∗
94		∗
95	27	FILE—READ—2.
96	27	MOVE SAL—2 TO PAR—SAL.
97	27	PERFORM SAL—COMP.
98	27	WRITE OUTREC FROM REC—2.
99	27	READ INFILE—2
100	28	AT END MOVE ALL HIGH—VALUES TO KEY—2.
100A	29	∗∗∗ EMPTY BRANCH for statement in line 99 ∗∗∗
101		∗
102	30	SAL—COMP.
103	30	IF PAR — „A"
104	31	ADD PAR—SAL TO SUM—SAL
105	31	ADD 1 TO N
106		ELSE
107	32	IF PAR—SAL>MAX—SAL
108	33	MOVE PAR—SAL TO MAX—SAL.
108A	34	∗∗∗ EMPTY BRANCH for statement in line 107 ∗∗∗
109	35	FINISH SECTION.
110	35	STOP RUN.

Total number of statements: 51
Total number of branches: 35

TABLE IV

DYNAMIC TEST PATH REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Testcase	Branch Start Statements	Date	Time
1	51 52, 53A, 55, 59, 62, 64, 65, 67, 83, 85, 88, 102, 104, 65, 67, 83, 87, 95, 102, 104, 100, 100A, 65, 67, 83, 85, 88, 102, 104, 65, 67, 83, 85, 88, 102, 104, 93, 93A, 65, 68A, 69, 75, 78, 100	12. 5. 1988	12:39:21
2	51, 52, 54, 55, 59, 62, 64, 65, 67, 83, 87, 95, 102, 107, 108, 65, 67, 83, 87, 95, 102, 107, 108, 100, 100A, 65, 67, 83, 85, 88, 102, 107, 108A, 93, 93A, 65, 68A, 69, 77, 78, 100, ∗ Error message: Assert violation	12. 5. 1988	12:41:19

TABLE V

PROGRAM COVERAGE REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Branch no.	Start stmt.	Total execution	Last execution	Not executed
1	51	3	1	
2	52	3	1	
3	53A	2	1	
4	54	1	0	
5	55	3	1	
6	58	1	1	
7	58A	1	1	
8	59	3	1	
9	61	1	1	
10	61A	1	1	
11	62	3	1	
12	64	3	1	
13	65	10	1	
14	67	7	0	
15	68A	3	1	
16	69	3	1	
17	75	1	0	
18	77	1	0	
19	78	2	0	
20	82	0	0	*
21	83	7	0	
22	85	4	0	
23	87	3	0	
24	88	4	0	
25	93	2	0	
26	93A	2	0	
27	95	3	0	
28	100	2	0	
29	100A	2	0	
30	102	7	0	
31	104	4	0	
32	107	3	0	
33	108	2	0	
34	108A	1	0	
35	110	2	0	

Total number of branches:	35
Number of branches executed:	34
Program coverage ratio:	97.14%

Dynamic Metrics:

Reliability = 0.33
Integrity = 0.11
Test Coverage = 0.79

TABLE VI

DATA COVERAGE REPORT

Module name: CMERGE Date: 12. 5. 1988 Page: 1

Data nr.	Lv. nr.	Data Name	Static Usage					Dynamic Usage				
			Prd	Arg	Res	Trn	Ini	Pre	Inp	Post	Out	N. A.
		FILE SECTION										
1	1	REC—1		a		t						*
2	2	KEY—1	p		r				in			
3	2	SAL—1		a					in			
4	1	REC—2		a		t						*
5	2	KEY—2	p		r				in			
6	2	SAL—2		a					in			
7	1	OUTREC			r	t						*
8	2	KEY—3									out	
9	2	SAL									out	
		WORKING—STORAGE SECTION										
10	1	W										*
11	3	SUM—SAL		a	r							*
12	3	MAX—SAL	p	a	r							*
13	77	N		a	r		i					*
14	77	PAR—SAL	p	a	r							*
15	77	DISP—RESULT		a	r							*
		LINKAGE SECTION										
16	1	LINK—DATA										*
17	2	PAR	p	a				pre				
18	2	RESULT		a	r					post		

Total number of data items (without Filler-s): 18

Number of predicates:	5	Number of PRE	asserted data:	1
Number of arguments	11	Number of INPUT	asserted data:	4
Number of results:	9	Number of POST	asserted data:	1
Number of transients:	3	Number of OUTPUT	asserted data:	2
Number of units:	1	Number of NOT	asserted data:	10

Data coverage ratio: 61.54%

The dynamic report generator produces the following reports:

— Test Log,
— Dynamic Test Path Report (see TABLE IV),
— Program Coverage Report (see TABLE V),
— Data Coverage Report (see TABLE IV).

Test Log contains everything which was displayed during the test run including all the assertion violations.

The Dynamic Test Path Report describes the test paths of each test case run executed by the Dynamic Analyzer. The report contains the test case number and the executed program path by printing the branch numbers in the same order as they were executed.

The Program Coverage Report is a table of program branches with their number of traversions since testing began. Branches which were not traversed are marked, this way new test data can be created to cover them, or unexecutable paths of the program can be revealed.

The Data Coverage Report describes the behavior of data items during the test run. It repeats some entries of the Static Data Usage Report to help the user to compare the static and dynamic results. The report describes the relation between the data items and the assertions. It indicates which data was input and/or output asserted or which data was not used at all.

The testing process using MICROTEST

The best known structural testing strategies are segment, branch and path testing. Segment testing requires each statement to be executed at least once during the test. Branch testing requires each branch (including the empty branches too) to be executed at least once. Path testing requires that all the paths in the program to be tested by at least one test case.

We chose the branch testing strategy because the segment testing is not effective enough [8] while the others require $O(n^2)$ [4] or more test paths in worst case, where n is the number of branches. (Branch testing requires $O(n)$ test paths.)

The process is the following: the programmer develops the COBOL program, and the tester develops the test program, independently. The tester selects test data from the specification. Then the Static Analyzer compiles the COBOL module, while the Assertion Compiler compiles the test program. The Dynamic Analyzer links the object codes and interprets it. If there are assertion violations, then either the COBOL or the test program should be corrected and compiled again.

In lack of assertion violations, the reports are investigated. If all the branches are covered then the testing is over, else new test data should be selected from the structure of the program. The new test program should be compiled again, and the process goes on until there are no assertion violations, and the branch coverage is acceptable.

Conclusions

We reviewed the development of test systems, and described the requirements of a high effective testing tool. The MICROTEST system is a result of these specifications. In the test data selection, both the functional and the structural methods are present.

The test data are separated from the program, so the test run can be repeated. At program execution, however, both the program and the test data are in a single interpretable binary program code. Since the Static Analyzer compiles the source code into the AL code, it is needless to instrument the source. Moreover, each module can be executed independently from the others since the assertion program simulates the calling and the called module. Thus bottom-up, top-down, and mixed testing strategies can be used as well.

The thorough evaluation of reports (especially the static and the coverage information) can significantly improve the program quality. The built-in debugger functions help program development.

The authors wish to express their gratitude to Mr. Harry Sneed, who has initiated the MICROTEST project and was the source of many ideas included in the system.

References

[1] M. V. Zelkowitz, Perspectives on Software Engineering, Computing Surveys, Vol. 10, No. 2, pp. 197—260, 1978.
[2] D. S. Alberts, "The Economics of Software Quality Assurance," Proc of National Computer Conference 1976, pp. 433—442.
[3] P. J. Smith, "The Requirement for Quality in the Design of Programming System," Proc. of Pragmatic Program and Sensible Software Conf., pp. 491—508, 1978.
[4] S. C. Ntafos, A "Comparison of Some Structural Testing Strategies," IEEE Trans. Software Eng., vol. 14. no. 6, pp. 868—875, June 1988.
[5] M. D. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of Structured Test Coverage Metrics," IEEE Trans. Software Eng., vol. 2. no. 2, pp. 80—85, March 1985.
[6] W. E. Howden, Functional Program Testing, IEEE Trans. Software Eng., vol. 6. no. 3, pp. 162—169, May 1980.
[7] W. E. Howden, "The Theory and Practice of Functional Testing," IEEE Software vol, 2. no. 5, pp. 6—17, Sept. 1985.
[8] W. R. Basili, R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies, "IEEE Trans. Software Eng., vol. 13. no. 12, pp. 1278—1296, Dec. 1987.
[9] J. C. Huang, "Program Instrumentation and Software Testing," IEEE Computer vol. 11, pp. 25—32, April 1978.
[10] T. J. McCabe, G. G. Schulmeyer, "System Testing Aided by Structured Analysis: A Practical Experience," IEEE Trans. Software Eng., vol. 11. no. 9, pp. 917—921, Sept. 1985.
[11] M. Majoros, H. M. Sneed "The Softests Program Test System," Systems and Software vol. 2. pp. 289—296, 1982.