Coverage Measurement Experience During Function Test

Paul Piwowarski
Mitsuru Ohba
Joe Caruso

International Business Machines Corporation

Abstract

This paper discusses the issues of test coverage measurement in industry and justifies the benefits of the measurement using a framework developed by the authors. Experience with the measurement is formalized and packaged so that other researchers in industry can share and reuse it. In the paper, function test of large-scale system software is defined and analyzed. Based on the discussions of function test, a framework for analyzing the function test error removal process is developed. An experience-based error removal model and a cost model are proven to be useful tools for justifying test coverage measurement during function test. Data obtained from a real project is analyzed using the framework for validation.

1.0 Introduction

Many papers have been published on test coverage measurements [2][6][8] . Most have been of a theoretical nature, discussing the benefits of various measurement criteria [11]. There has been little discussion of the successful use of test coverage measurements on large projects, and whether it has proved to be a cost effective method for either locating errors or assuring quality (e.g., reliability, maintainability) of a product [14].

One reason for the lack of data on test coverage measurements during software testing is the difficulty of getting test coverage numbers on large complex products [5]. The academic community may enjoy discussing the merits of various test coverage measures. The user community would like to get any useful coverage numbers at all, using any measurement criteria.

Within IBM, statement and branch coverage measurements have been done on some very large projects, such as operating systems and compilers using internal IBM tools since the late 1960s. The first project to seriously measure code coverage was a new operating system for a small business computer in the early 1970s at IBM Rochester, Minnesota. A hardware tool was developed to measure the operating system's statement and branch coverage. Before this project, we believed that 99 per cent statement coverage and 95 per cent branch coverage were generally achievable. We found that this was not the case.

In the late 1970s, another hardware tool was built at IBM San Jose, California. The tool was developed to measure test coverage of an I/O subsystem of an operating system for the IBM System/370 machines. At the same time, a group at IBM Poughkeepsie, New York, developed a software tool to measure test coverage of an operating system kernel for the IBM System/370 machines. This tool proved that a software tool could do the same thing that its expensive predecessors did.

From the late 1970 to the early 1980s, test coverage measurements were done on some large projects such as operating systems. Through those experiments, we found that test coverage does not directly correlate to actual reliability of those products. Users of the operating system sometimes found more errors in parts whose test coverage numbers were relatively higher than the other parts. From the users' point of view (mean time to failure) we found stronger correlation between usage of the operating system and MTTF than between a test coverage number and MTTF.

A software test coverage measurement tool called EXMAP (Execution Time Mapping Tool) was developed to measure OS/MVS system code at IBM Poughkeepsie in the late 1980s. EXMAP tracks test coverage with minimum overhead while the tester is

0270-5257/93 $03.00 © 1993 IEEE

running test cases in a normal testing environment. The tester does not have to test the product using a debugger, or recompile the code to insert hooks into the modules.

The tool has been used by testers on some large projects since it was developed. We have learned that measuring test coverage does take system and human resources. Therefore the following issue has been raised by testers: "If you increase coverage by the use of the tool, do you really find more errors, and is the savings realized in fewer errors worth the cost?"

Test coverage measurement provides feedback to product developers which will encourage them to increase coverage by adding additional test cases. This increased coverage will, in turn, result in lower error removal costs as well as fewer errors shipped in the product. Therefore, any assessment of value will require that a model of the relationship between test coverage and product quality be developed.

The results in this paper deal with the use of statement coverage during function testing. The same discussion can also be applied to unit testing. Other measurement criteria other than statement and branch coverage (e.g., data-flow coverage) may be also cost effective.

In this paper, we first define what function test is in our environment in terms of its goals, process, and measurements. We define the function test problem and then describe how it is solved in practice. We describe our tool and the state of test coverage measurement in industry. We then analyze the relationship between test coverage and error content of a product. We develop, based on our experience, our error removal model and cost model. We validate the error removal model by analyzing data taken from an experimental project. Our conclusions, and directions for future research are presented.

2.0 Testing of Large Scale Software

The observations in this section are based on a study of testing of large scale software in IBM conducted by the authors in 1991 [12].

The test process consisted of three phases:

1. Unit test
2. Function test
3. System test

Unit test was done by the programmers who wrote the code being tested. Function test was usually done by a different group within the same development organization. System test was generally done by a group that was independent of the development organization.

2.1 Overall Process

Testers generally thought that the goal of testing was to remove as many errors in a product as possible during the given development cycle. Some testers felt that the goal of all testing was to assess quality of a product from the customers' point of view.

The number of errors found during unit, function, and system tests was the basic measurement for these test phases. A hypothetical error profile through the development phases is shown in Table 1. The assumptions are that 47.37% of errors are injected during design, 52.63% are injected during coding, and no errors are injected during the test phases. (See Stott [15] and Jones [7] for a discussion of error removal profiles.)

Table 1. Hypothetical Error Profile by Phase

Phase	Errors Entering	Injected	Portion removed	Remaining
Design	.00	47.37	.80	9.48
Code	9.48	52.63	.50	30.92
Unit Test	30.92	.00	.50	15.46
Function Test	15.46	.00	.60	6.18
System Test	6.18	.00	.45	3.40

2.2 Unit Test

The goal of unit test was to remove as many errors in a module as possible and to know that the module under test met its low level design (or module level specifications).

Unit test was done by the programmer. Testing was white-box or "structural" and the process varied from developer to developer. There were several techniques used to generate test cases. Test cases generated and problems discovered during unit were occasionally recorded.

Although many viewed the exit criteria from unit test as a certain form of test coverage (e.g., 100% statement coverage), this coverage was only occasionally measured. Alternatively, there were cases in which a successful run of an acceptance test suite of a function test was considered the formal exit of a unit test. Normally the programmer determined when unit test was complete based on the tester's experience with the module.

The number of errors reported was the typical measurement for unit test. One project also did some statement coverage measurement.

2.3 Function Test

The goal of function test was to test the functions listed in the functional specifications (black-box testing) and remove as many errors in the code as possible before system test.

Testing was black-box or "functional" testing. Some function testers looked at code in order to make sure that test cases hit important areas of the code. One project had been experimenting with random testing and trying to automate it. The test plan and test cases were documented and reviewed. The problems found during function test were recorded.

The entrance criteria was the availability of the code that implemented the function. In some cases, a subset of function test cases were run as an acceptance test. Exit criteria was essentially the successful completion of a set of function test cases.

Measurements were errors found and number of test cases executed. Some projects measured statement coverage and used that as a measure of testing effectiveness.

2.4 System Test

Some testers felt that the goal of system test was to find errors under realistic or stressed environments. Others felt that the goal was to integrate products in customer-like environments and assess quality of a product under test.

Testing was the black-box or "functional testing." There were cases in which other products which were developed outside development organizations (e.g., different divisions) were also integrated into a test environment and tested. Test cases developed for function test were sometimes selected and used for system test. The test plan and the test cases were documented and reviewed. The problems found during system test were consistently recorded.

Entrance criteria was availability of the code that implemented the functions, not the entire system that implemented all the functions specified in the functional specifications. Some projects ran an acceptance test case suite.

The frequency of failure occurrences and types of errors found in the late stage of system test were analyzed to determine if a product was ready.

3.0 Characteristics of Function Test

3.1 The problem

What function testers try to do usually is to verify that a product under test properly generates outputs for given inputs as they are specified in the functional specifications. If an output generated by the product does not satisfy the functional specifications, testers say: "I found an error."

This function test problem is formulated as follows. Suppose W is an input space such that members of W are states of the input vector w, S is a system state space such that members of S are states of the system state vector s, and Y is an output space such that members of Y are states of the output vector y. A set of functional specifications which defines a mapping from the input space to the output space can be regarded as a discrete function $f: W \times S -> Y \times S$, where W, S and Y are finite sets.

For convenience, the Cartesian products $W \times S$ and $Y \times S$ are denoted by X and Z respectively. An element of X, x, is a state of vector (w, s), and an element of Z, z, is a state of vector (y, s). Let N be the size of space X (N = |X|), and K be the dimension of vector x ($K = \dim(w) + \dim(s)$). X is a set of combinations of possible input values and internal states of a program.

The size of the input space N is bounded, and the upper bound is given by:

$$N \leq O(\exp(K)).$$

(Proof) The Cartesian product of input space W and state space S is the largest set of the combinations of w and s. If all the variables of x and s are independent, the number of possible combinations of vector x, i.e., N, is $O(\exp(K))$.

The problem which is $O(\exp(K))$ is not practically solvable. It is generally infeasible to find the optimum solution for a given problem. If we can try all the possible cases, we can find the solution. Therefore, the testers' problem is to find a partial solution which satisfies a certain set of conditions in a given period of time.

The problem to be solved is to find set X′ such that X′ is a subset of X and its size is "feasibly small." X′ is a set of test cases. If "feasibly small" means O(K), the problem can be reformulated by the following proposition:

Proposition: for given X, Z and $f: X -> Z$, there exists subset X′ of X such that X′ satisfies the following conditions:

1. $| Z | < | X' | < O(K)$,
2. $Z' = Z$ where f: $X' -> Z'$.

3.2 Solution in practice

A typical approach taken to find X′ by testers in practice is:

1. if constraints on the input space are strong and the size of X is "feasibly small," select X as X′;
2. if constraints on the input space are not strong so that the size of X is not "feasibly small" but considerably smaller than the upper bound, 1) select X″ so as to satisfy the second condition of the proposition; 2) if the size of the X″ is significantly smaller than the size of X, add some elements of X to X″; 3) if X″ becomes reasonably large, select X″ as X′;
3. if the input variables are independent and the size of X is close to the upper bound, select X* so as to satisfy only the second condition of the proposition where $|X^*| = |Z|$

The first case is that functional specifications are simple so that testers can try all the possible cases. The second case is that functional specifications are fairly complex so that testers cannot try all the possible cases. Testers analyze the given functional specifications, identify classes of inputs and outputs, and try to exercise all the relations between input and output classes. The last case is that functional specifications are too complex to test systematically. Testers do not try to analyze a structure of the functional specifications. Testers try to show either that the product works correctly (optimistic testing) or that the product does not work in some cases (pessimistic testing).

There are no consistent algorithms used to find X″ and X* in practice. X″ can be for example obtained using the equivalence partitioning method. Testers sometimes use their informal (personal) techniques which can be seen as equivalence partitioning. The most consistent way of finding X″ is random search, though it requires a significant amount of test cases. Another way often used for finding X″ in practice is: to fix values of some input variables to reduce the search space.

A common way of finding X* is to select some elements of X so that X* covers typical elements of X based on the usage of a function. The most optimistic way of selecting X* is to try a small set of inputs which are expected to be the most typical cases. The most pessimistic way of selecting X* is to try all the inputs that have caused failures in the past. These can be interpreted as an informal way of finding X*. X* in these cases may not be complete to cover Z.

The most important condition to determine if designed test cases are sufficient is the question: "Are all the possible output patterns covered by test cases?" If this condition is satisfied, the next condition is the question: "Is the number of test cases reasonable?" Testers sometimes refer to their experiences to answer this question. The typical number that is frequently referred is "1 test case per 10 lines of code."

To ensure that test cases for function test are properly designed and completely cover Z, testers sometimes refer to a test coverage index of their test cases. If test cases are complete, each test case exercises a part of the program which is specifically exercised by the test case as well as common parts which are also exercised by other test cases. Therefore, for each test case there exists a part of the program which uniquely corresponds to the test case. If test cases are complete and adequate to cover X″ or X*, they must exercise all the parts of the program (i.e., 100 percent test coverage).

3.3 Function test and test coverage

By assuming that test cases are randomly selected from space X so as to satisfy either |X*| = |Z| or |X'| > |Z|, the probability that a test case exercises k new blocks which have not been exercised after covering n blocks more than once is given by the hypergeometric distribution [16]:

$$Prob\{k|n\} = \frac{{}_{N-n}C_k \times {}_nC_{p-k}}{{}_NC_p},$$

where N is the number of blocks in a program, and p is the average number of blocks covered by a test case during function test. $p - k$ out of p blocks have already been covered by other test cases.

The expected number of blocks newly exercised by the i-th test case is given by:

$$l_i = \sum_{j=1}^{p} j \times Prob\{j|n_{i-1}\},$$

where n_{i-1} is the number of blocks covered before running the i-th test case. n_i is recursively estimated using l_j as follows:

$$\bar{n}_i = \sum_{j=1}^{i} l_j,$$

where $l_1 = p$, and $\bar{n}_1 = p$. Therefore, we obtain:

$$l_i = p \times \frac{N - \bar{n}_{i-1}}{N} = \frac{p}{N}[N - \bar{n}_{i-1}].$$

Since $\bar{n}_i$ can be regarded as an integration of l_i, we can formulate a continuous approximation function:

$$\frac{d}{dx} n(x) = \frac{p}{N}[N - n(x)],$$

where x is the number of test cases executed. By solving this equation, we obtain:

$$n(x) = N(1 - e^{-\frac{p}{N}x}).$$

$n(x)$ is the function that describes the relationship between the number of blocks covered and the number of test cases executed. By dividing $n(x)$ by N, we obtain the function that describes the relationship between coverage and the number of test cases:

$$c(x) = 1 - e^{-\frac{p}{N}x},$$

where $c(x)$ is coverage after executing x test cases. The function suggests that increasing test coverage beyond a certain point is not cost effective. The function also suggests that some of the blocks in a program are frequently exercised while some others are rarely exercised.

If we assume that errors in blocks which are frequently exercised are more likely to be detected and removed, then we can expect that those frequently exercised blocks are more likely to be less error prone and more reliable than other blocks. We can also expect that a coverage value is not necessarily equal to the ratio of error free code though it is related to reliability of code (a higher coverage value implies higher reliability of code).

If we assume that errors in a program (or a part of a program) are homogeneously distributed and testers detect errors at the time when errors are sensitized the first time, then we can expect that reliability growth of function test in terms of the number of errors found is exponential (e.g., Goel-Okumoto's NHPP model).

4.0 Test Coverage in Industry

In the survey of IBM testing organizations previously mentioned, we found that testers:

- were familiar with test case coverage measures,
- believed that the use of coverage measures would help them find errors and improve the quality of their products,
- but generally did not use test case coverage tools.

Testers did not use coverage tools, not because of a lack of knowledge of them, or lack of belief in their worth, but because coverage tools had proved to be to difficult to use.

A number of test case coverage measurement tools were available. However the available tools could not be used for large projects, system code, or function test in general. The reasons for this were as follows. The code to be tested was run under control of a debugger, and could not be tested in its normal function test environment. The code measurement process increased the execution time of the tested programs beyond acceptable limits. The measurement tools did not support the languages used for specific projects. The effort to do the measurement (setup time, special steps such as recompilation of code, etc.) was too large for resource constrained testing organizations.

Some of those tools, for example, used the history file generated while a debugger was monitoring the code. Some of them did not support assembler and the IBM internal system programming languages. The overhead in added execution time of the program under test, and creating the history file was sometimes very large.

Despite the good intentions of function testers who wanted to measure their coverage, they often did not have an accurate measure of their coverage. Testers often ended up making their own estimates of code coverage.

4.1 Tool for measuring system code

To measure the coverage of large system code such as operating systems, compilers, etc., L. Balfour of IBM Yorktown developed an internal test coverage measurement tool called Execution Time Mapping Tool (EXMAP). The tool uses measurement techniques that allow the tested programs to run in their normal user environments, and not under control of a debugger. The tool supports the languages that are used in IBM for large project development and systems level code: System/370 Assembler, C, and IBM internal high-level languages. The overhead in terms of additional execution time is small. Testers have found that the execution time of their programs is about 10% greater when the tool is monitoring coverage.

EXMAP has been used within IBM to measure statement and branch coverage of operating system code, compilers, and other complex products. The tool provides a summary of test coverage for each function (in C) or procedure (for PL/I based system programming languages), and also annotated listings showing the execution status of each statement. The tester can have reports of test case coverage for an individual test case, or cumulative results for a group of test cases. See Appendix B for an example of a summary report and for an example of an annotated C listing.

With EXMAP, testers can measure both statement and branch coverage during unit or function testing of operating systems (e.g., MVS, VM/CMS), except for deep system code (e.g., IPL modules) To measure test coverage of deep system code, the tool has been enhanced to work with a System/390 hardware simulation tool. With this enhancement, programmers and testers can measure any part of an operating system.

As testers became more familiar with EXMAP and test coverage measurements, they have wanted functional enhancements of the tool. Typical requirements from the users were: 1) the graphic presentation support of measurement results, 2) integration of the measurement capability into the debugging environment that consists of the hardware simulation tool and a source level debugger, 3) a method to automatically select regression test cases based on test case coverage of changed modules, 4) new coverage measurements (e.g., data flow coverage [13]), coverage on platforms other than S/390 (e.g., RS/6000, PS/2). EXMAP is continuously being enhanced based on these user requirements.

4.2 What we have learned about test coverage

Although test coverage had been measured on some projects in IBM since the early 1970s, the experiences of the testers were not consistently recorded. After our review of testing in IBM, we recommended that experiences be centrally recorded and shared by all of our testers [1]. We started collecting information about test coverage measurements in 1991. We also have planned experiments on coverage measurements to enhance our understanding of the value of test coverage measurements in testing.

When test coverage had not previously been measured, testers tended to overestimate coverage of their test cases. The first time testers measured coverage during function test, they found that the coverage was in the range of 50% to 60%. The testers were surprised at the low percentage of coverage they were getting. They expected a much higher percentage of code coverage. Some testers estimated that their coverage was 90% or higher.

Once coverage reports were available, testers were able to design new test cases, and improve existing ones so that coverage reached the range 70% to 80%. Increasing coverage beyond this range proved to be difficult. These are some of the reasons. There is code that can only be tested using special hardware available during system test, but not during function test. Some code cannot be reached. This may point out code that was no needed, but there are cases (described below) where unreachable code may be left in a product. Some conditions are very difficult to create, and have a low probability of occurrence. The tester may decide that it is not cost effective to test these conditions.

Typical examples of unreachable code are found in the following cases. The program may have checks for "impossible" error conditions. For example, in a case statement, there may be cases for all conditions, but a default case is still provided in the event a future program change deletes one of the cases. Code may be included for features that are not part of this release. The project may have standard libraries included in the program (for example I/O routines) that provide multiple functions, and the program is only using some of the functions.

Based on our survey of testing in IBM, and the analysis done in "Characteristics of Function Test" (above), we have concluded that:

- 70% statement coverage is the critical point for our function test to assure that test cases sufficiently exercise and cover all the output patterns (the conditions of the proposition),
- 50% statement coverage is generally insufficient for our function test to assure that test cases exercise and cover all the output patterns (the second condition of the proposition),
- beyond a certain range (70%-80%), increasing statement coverage becomes difficult and is not cost effective.

From actual measurements of code coverage on system products, 70% statement coverage can be achieved during function test.

5.0 Relationship Between Coverage and Errors Found

5.1 Coverage-based reliability growth model

We formulate the relationship between increase of statement coverage and decrease of remaining errors in a product using a simplified software reliability growth model. Software reliability growth models are mathematical functions which describe relationships between the number of errors found during test and the amount of testing (e.g., time, test runs).

The amount of testing is measured in terms of the number of runs of test cases. To find errors in a part of code, test cases should sensitize (traverse) the part of code at least as many times as the number of errors that test cases detect. If we have found three errors in a particular piece of code, we can say that test cases sensitized the particular part of code more than three times.

Here we assume that pieces of code of a product are grouped based on the level of sensitization frequency. Code is sensitized at least the number of times that test cases detect errors (i.e., the number of errors detected). Suppose T is the number of test cases which exercised a particular group of code, and m is the number of errors detected by the test cases, the sensitization level of the code group should satisfy the following:

$$s = dT, \text{ and } m \leq dT,$$

where d is a constant ($0 < d \leq 1$).

We can describe the relationship between the sensitization level and the number of errors found (see Appendix A):

$$m(s) = M(1 - (\frac{1}{K})^s),$$

where m(s) is the number of errors found up to the time when a part of code is sensitized at frequency level s, K is a error detection rate constant, and M is the number of errors initially in the part of code. By normalizing (dividing m(s) by M), we obtain the error removal ratio:

$$r(s) = 1 - K^{-s}.$$

K primarily depends on the number of errors remaining in the code under test. K also depends on the skill level of testers who design test cases and analyze test results. Generally K is determined by:

$$K = \frac{b}{q} + 1,$$

where b is a constant which is determined based on historical data, and q is the error density (the number of errors per unit size of code).

5.2 Framework for error content analysis

The following is a summary of observations which we had learned from the experience:

- We remove approximately 60% of errors in a product during the first session of function test.
- We remove approximately 60% of remaining errors in a product by running a set of test buckets once. We can remove approximately 10% more errors by running an improved set of test buckets again (by changing test cases in the buckets or changing order of running test cases).
- If we run the same set of test buckets in an exactly same order, we cannot find and remove new errors.
- We usually cover 50% of code during the first session of function test. This implies that errors are not homogeneously distributed.
- We can increase statement coverage by 10% by knowing the coverage and improving test cases.

The observations indicate that the first 50% of code (five groups) must contain more than 60% of errors in a product. It implies that the error distribution among code groups is not homogeneous.

We found the following geometric distribution of errors satisfies our observations:

$$P(i) = a(1 - a)^{i-1}.$$

where i is the code group, P(i) is the error density for group i, and a is the distribution parameter $(0 < a < 1)$. If L is the number of code groups, it is assumed that a is greater than 1/L. If we divide the code under test into 10 groups, a should be larger than 0.1. For example, if $a = 0.2$, the most error prone code group contains 20% of errors of a product instead of 10% (homogeneous distribution of ten groups).

By assuming that the sensitization frequency is proportional to error content of each code group, we obtain Table 2. In Table 2, each row represents a sensitization level, where j is the sensitization level that equals i, M is the number of errors, and n is the sensitization level that is greater than the number of code groups. There is a case in which same groups of code are sensitized by test cases without sensitizing a new group of code. The (j + 1)th row represents the case. In general, the number of defects removed up to the time when the 1st to the l-th group are covered and the 1st group sensitized n times is given by:

$$m(l) = M\sum_{i=1}^{l} P(i) \times (1 - K^{-(n-i+1)}),$$

where $n \geq l$.

Based on our historical data, the error distribution parameter (a) is typically 0.2, and K is usually in the range between 1.5 and 3. $1.7 \leq K \leq 2.0$ is said to be realistic and typical in our environment.

In terms of error content, if we assume that $K = 2$ and $a = 0.2$, 67.2% of errors are exposed during the first session of function test that covers the first 50% of code (group 1 through 5), and 57.3% of errors are removed. This is consistent with the first observation.

If we add a new set of test cases to increase statement coverage by 10%, group 6 will be exposed and 50% of remaining errors in groups 1 through 6 will be removed. If we assume $K = 2$ and $a = 0.2$, 73.8% of errors will be exposed by covering the first 60% of code, and 8.2% more errors will be removed. 65.6% of errors will have been removed at the end of the phase. This is consistent with the second observation.

6.0 Value of Test Coverage Analysis

6.1 Experience-based error removal model

Although simplistic, the following assumptions are necessary for the development of a value model:

1. that test buckets will sensitize clusters or groups of statements as a result of execution,
2. that the error distribution among groups is not homogeneous,
3. that the error removal process is best described by the error profile discussed in the section on testing of large scale software,
4. that costs can be assigned to removing errors at each phase of development and that these costs increase at later phases of development,
5. that testing done with a priori knowledge of the distribution of errors and that the groups with highest error density are tested first,
6. that testing involves repetitive sensitization of these same groups plus new groups of statements

Table 2. Error distribution, sensitization and error removal

group #	1	2	3	...	i	...
Errors	aM	$a(1-a)M$	$a(1-a)^2M$		$a(1-a)^{(i-1)}M$	
1	$aM(1-1/K)$					
2	$aM((1-(1/K)^2)+(1-a)(1-1/K))$					
3	$aM((1-(1/K)^3)+(1-a)(1-(1/K)^2)+(1-a)^2(1-1/K))$					
.						
.						
j	$aM((1-(1/K)^j)+(1-a)(1-(1/K)^{(j-1)})+\dots\dots\dots+(1-a)^{(i-1)}(1-1/K))$					
j+1	$aM((1-(1/K)^{(j+1)})+(1-a)(1-(1/K)^j)+\dots\dots\dots+(1-a)^{(i-1)}(1-1/K)^2)$					
.						
.						
n	$aM((1-(1/K)^n)+(1-a)(1-(1/K)^{(n-1)})+\dots+(1-a)^{(i-1)}(1-(1/K)^{(n-i+1)}))+\dots$					

which were not previously sensitized (repetitive testing with expanding test buckets).

Figure 1 shows an actual error distribution for a product with 10 code groups of roughly equal size.

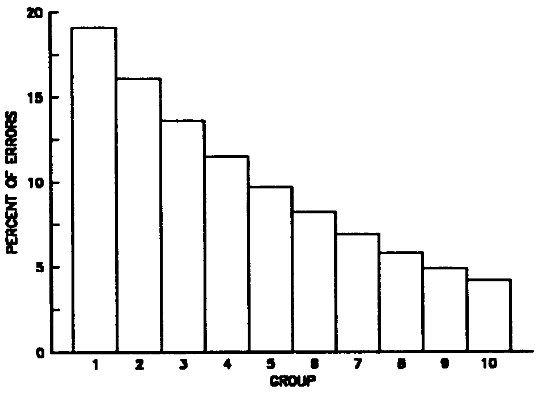

Figure 1. Group Error Density for a Product

Using the above error density, a hypothetical model can be constructed which illustrates the relation between code coverage and product quality. Assume that the first test bucket run sensitizes group 1. When the test bucket is run a second time it is augmented with new test cases and sensitizes groups 1 a second time and group 2 for the first time. The third time it is run it sensitizes groups 1, 2, 3 and so on. Also assume that K, the error detection rate constant, is 2 so that 50% of the remaining errors are removed with each sensitization. This would lead to the pattern of errors discovered in Table 3. The cumulative error removal ratio in the right hand column of Table 3 can be plotted against the number of code groups to establish a hypothetical relation between code coverage and number of errors removed. The resulting relation can be viewed in Figure 2.

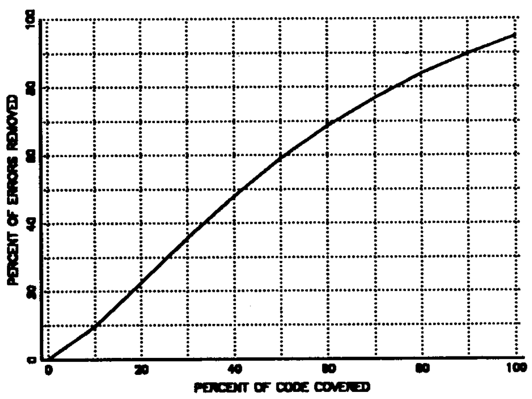

Figure 2. Example of Error Removal Rate and Code Coverage Relationship

6.2 Cost Model

Given the relation between the number of errors removed and code coverage displayed in Figure 2, we see that the 60% Function Test error removal rate in the error removal model in the previous section would imply statement coverage of roughly 50%. Figure 2 also shows that a 10% improvement in coverage would lead to approximately a 70% removal rate. If we plug 70% back into the error profile for Function Test (Table 1 revised to Table 4), the percent of errors shipped drops from 3.4 to 2.55. This is a quality improvement of 25% from the original model.

Table 3. Example of an Error Discovery Pattern

Sensitization	Percent of Errors Removed for Group i										
	$i=1$	2	3	4	5	6	7	8	9	10	Total
1	9.6										9.6
2	4.8	8.1									22.4
3	2.4	4.0	6.8								35.6
4	1.2	2.0	3.4	5.8							47.9
5	.6	1.0	1.7	2.9	4.9						59.0
6	.3	.5	.9	1.4	2.4	4.1					68.6
7	.1	.3	.4	.7	1.2	2.1	3.5				76.8
8	.1	.1	.2	.4	.6	1.0	1.7	2.9			83.9
9	.0	.1	.1	.2	.3	.5	.9	1.5	2.5		89.8
10	.0	.0	.1	.1	.2	.3	.4	.7	1.2	2.1	94.9

Table 4. Example of a Modified Error Profile by Phase

Phase	Errors Entering	Injected	Portion removed	Removed	Remaining
Design	0.00	47.37	.80	37.89	9.48
Code	9.48	52.63	.50	30.92	30.92
U/T	30.92	.00	.50	15.46	15.46
F/T	15.46	.00	.60 > .70	10.82	4.64
S/T	4.64	.00	.45	2.09	2.55

Furthermore, if we assume that the relative costs of removing errors are 1, 3 and 20 for Function Test, System Test and after shipment, respectively, we can see from Table 5 that the overall cost of removing errors has been reduced. (See Boehm [3] for a discussion of relative costs of removing errors.)

Table 5. Example of Improvement in Cost of Removing Errors

Phase	Relative Cost	Removal Cost		Percent Reduction
		Before Analysis	After Analysis	
Function Test	1	9.28	10.82	-17
System Test	3	8.34	6.27	25
Total Internal		17.62	17.09	3
After Shipment	20	68.00	51.00	25
Total		85.62	68.09	20

Although function test error removal costs have increased, the decrease in system test costs more than offset the increase so that total internal test costs decrease by 3%. Of course most of the reduction in error removal costs are realized as a result of fewer errors in the product after shipment since these errors are expensive to correct. The total reduction for error removal costs for this example is 20%.

7.0 Model Experience

7.1 The project and observations

In order to test the assumptions previously stated, test coverage analysis data was collected from a large development project. The program analyzed was 780 KLOC of "deep systems" code written in an internal system programming language from a large operating system. The function test team worked continuously for one and a half years on this test. Test cases were developed both manually and automatically using random combinations of inputs. Approximately 90 KLOC of code was added after starting this function test.

The testing process for this project was significantly different from others. Once unit test was done, function test and system test were done completely in parallel by two groups. Usually system test is done after the completion of function test. The different process is possible because the goals of function and system tests are different. To ensure that all the function needed for the system test was available, a subset of function test cases was selected and used as an acceptance test for the function and system tests. One of the exit criteria for unit test was successful completion of all the acceptance test cases. This significantly reduced the number of errors found during the function test.

The data is displayed in Table 6. During this test, statement coverage was measured and reported periodically (usually once a week). The measurement was done during periodic regression runs to verify corrections for errors found during the period. All the test cases that had caused failures during test runs were included in the regression test buckets (suites of test cases). All the test cases in the regression test buckets were automatically executed during the periodic regression run. The cumulative number of test cases shown in Table 6 is the size of the total regression buckets (not the number of all the test cases used for testing). The data for Period 1 represents the test coverage and errors removed prior to test coverage analysis. The subsequent periods reflect increased test coverage and errors removed as a result of test coverage analysis. The length of Period 4 was as twice long as the other three periods.

Table 6. Test Coverage Data for a Portion of a Large Operating System

Period	Cumulative Coverage	Errors Found	Cumulative Test Cases	
			Total	Random
1	64	100	420	0
2	68	90	540	70
3	69	70	900	180
4	69	80	1460	530

7.2 Analysis of the error removal process

As discussed earlier, the level of sensitization is proportional to the number of test cases executed. In order to determine the level of sensitizations during each period of time, we assume that each sensitization consisted of 100 test cases, e.g., 4 sensitizations were assumed for Period 1 since there were 420 test cases. We also assume that the first sensitization included approximately 34% of the code, the second sensitization included an additional 17% of the code, the third sensitization included an additional 9%, and the fourth sensitization included an additional 4% to make a total of 64% coverage.

Another assumption was required for the distribution of errors in the code since this data was not available. It was assumed that the data followed a truncated geometric distribution ($.01x.99^{0,1,...,99}$) with each interval representing the error content of 1% of the product. The intervals were then combined to obtain error densities for each sensitization.

Finally, it was assumed that the total 340 errors removed represented 70% of the errors coming into function test. This assumption is consistent with the modified error profile presented earlier.

Table 7. Sensitization Model of Operating System Data

Sensitization	Percent of Errors	Cumulative Coverage	K	Percent Errors Removed	
				Predicted	Actual
1	23.4	34	10.00		
2	20.0	51	10.00		
3	17.0	60	10.00		
4	14.5	64	10.00	22	21
5	3.3	68	3.33	39	39
6-15	.8	69	6.66	71	70

The results of the analysis are in Table 7. Table 7 shows that the sensitization model can fit the observed data well when comparing the predicted to actual percent of errors removed. However, this fit is accomplished by varying the value for K between the periods.

The value for K is extremely high for the first 4 sensitizations (Period 1). A possible explanation would be that the test buckets for Period 1 consist largely of test cases which have already been run in unit test. Also, it might be that the code that is normally tested without coverage analysis (during unit test and the first phase of function test) is better understood and therefore of higher quality than code which is exercised only after coverage analysis.

The fifth sensitization (Period 2) increases coverage from 64 to 68% and exercises code statements not normally tested. Here K has a much lower value which implies that the first set new test cases as a result of code coverage analysis have uncovered a significant number of new errors.

Finally, in going from 68 to 69% (Period 3 and 4), the code is sensitized 10 times which implies difficulty in going beyond 69% coverage and a "scrubbing" of code that has already been covered. This additional scrubbing of the code results in fewer and fewer errors removed each sensitization and K rises back up to 6.66 since the code covered is of higher quality after the fifth sensitization.

7.3 Conclusions

Some results of experiences using test coverage measurement were discussed. We believe that sharing our experience with other people in industry and in research is one of the keys to improving our software testing practice and to deepen our knowledge of software testing. Here we discussed that measurement of statement and branch coverage of large system software can be done, and is cost effective in removing errors.

Through the use of a test coverage tool designed to be used while a product is run in its normal environment (rather than under a debugger), testers in IBM have shown that it is possible to measure test coverage during function test. It is then possible to design new test cases and improve existing ones to increase test coverage. We have shown that it is cost effective to spend the human and system resources to improve test coverage. The savings in the errors found because of the increased test coverage more than compensates for the cost of the resources needed to measure test coverage.

An error removal model was developed. The application of the sensitization model to actual data highlights the fact that K (the error detection rate constant) cannot be regarded as static throughout the testing process. Furthermore, the number of sensitizations assumed will have a significant impact on the predicted error removal ratio. There are many other assumptions upon which the model is based that may raise questions. However, the model seems to provide a reasonable framework by which to explain the error removal ratio throughout function test.

The test case coverage results discussed here were done during function test of large system software products. Test coverage metrics can be applied to unit testing of the modules of a product. Typically the unit test process varies considerably among developers and test effectiveness is only occasionally reported. If the unit tester has a good tool available, the tester can prove what has been tested. Although it is not feasible to reach 100% statement coverage in function test, it is possible to do that in unit test. The unit tester has access to stubs, debuggers, etc., to exercise paths that are difficult or impossible to test during function test. If a good test coverage measurement tool is available, an exit criteria of unit test can be 100% statement coverage.

Recently data flow coverage measures have attracted attention. The test coverage measurement tools should be enhanced to provide data flow coverage metrics. It remains to be seen how much these data flow metrics will aid in the development in test case design above what can be done by using statement and branch coverage metrics in large products.

Areas of work in IBM and industry are the use of test coverage measures during unit test, building an experience-base for test coverage measurements, and experimentation in the area of data flow coverage measurements.

7.4 Acknowledgements

We thank V. R. Basili of University of Maryland for his contribution to our study of IBM software testing and for his advice on test coverage measurement experiments. We were influenced by his ideas on the experience factory.

We thank J. McDonnell, M. Mundy and other colleagues of IBM Poughkeepsie for their comments, participation and encouragement for this study. We have learned many things from them.

And we thank our management team. Without their understanding and support, we could not have done the study. Especially we would like to express our personal appreciation to our friend, a colleague and a member of our management team, J. C. Culbertson of IBM Lexington, who suddenly passed away during the study.

7.5 References

[1] Basili, V. R., and Caldiera, "Methodological and Architectural Issues in the Experience Factory", Proc. of the 16th Annual Software Engineering Workshop (SEL-91-006, NASA GSFC), Maryland, December 1991, pp. 29-46.

[2] Basili, V. R., and Selby, R. W., "Comparing Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987 pp. 1278-1296.

[3] Boehm, B. W., "Software Engineering Economics", Prentice-Hall, New Jersey, 1981.

[4] Currit, P. A., Dyer, M., and Mills, H.D., "Certifying the Reliability of Software", IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986.

[5] Gelperin, D., Hetzel, W., "The Growth of Software Testing", Communications of the ACM, Vol. 31, No. 6, June 1988, pp. 687-695.

[6] Huang, J. C., "An Approach to Program Testing", ACM Computing Surveys, September 1975, pp. 113-128.

[7] Jones, C. N., "Programming Productivity", McGraw-Hill, New York, 1986.

[8] Miller, E. F., "Program Testing: An Overview for Managers", COMSAC, 1978.

[9] Musa, J.D., Iannino, A., Okumoto, K., "Software Reliability, Measurement, Prediction, Application", McGraw-Hill, New York 1987.

[10] Ohba, M., "Software Reliability Analysis Models", IBM Journal of Research and Development, Vol. 28, No. 4, July 1984, pp. 428-432.

[11] Ohba, M., "Software Quality = Test Accuracy x Test Coverage", Proc. of 6th ICSE, Tokyo, September 1982, pp. 287-293.

[12] Ohba, M., and Basili, V. R., "A Study of Large Scale Software Testing", Proc. of the 16th Annual Software Engineering Workshop (SEL-91-006, NASA GSFC), Maryland, December 1991, pp. 199-207.

[13] Rapps, S., Weyuker, E., "Selecting Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April 1985, pp. 367-375.

[14] Stahl, W., "Packing Your Testing Toolbox", Computerworld, October 9, 1989, pp. 83-89.

[15] Stott, D. R., "Improving the Quality of Critical Decisions", ASQC Quality Congress Transactions, San Francisco, May 1990, pp. 518-523.

[16] Tohma, Y. et al, "Structural Approach to the Estimation of the number of residual software faults based on the Hyper-Geometric Distribution," IEEE Trans. Software Engineering, Vol. SE-15, No. 3, 1989, pp. 345-355.

Appendix A: Coverage Based Reliability Growth Model

It is reasonable to assume that new errors are captured if and only if new blocks of a program are exercised the first time, because test team members put their focus on the segment or function tested by a test case. If we assume that errors are distributed homogeneously, we obtain:

$$\frac{d}{ds} n(s) = \frac{s}{N} [N - n(s)],$$

$$\frac{d}{ds} m(s) = \frac{M}{N} \times \frac{d}{ds} n(s),$$

where $n(s)$ is the function that describes the relationship between the number of blocks covered and the number of sensitizations, N is the number of blocks in the program, $m(s)$ is the function that describes the relationship between the number of errors found up to the time when the program (or the part of the program) has been sensitized s times, and M is the number of number of errors in the code.

By solving the simultaneous differential equations with respect to $m(s)$, we obtain:

$$m(s) = M(1 - e^{-\frac{p}{N}s}).$$

This $m(s)$ is isomorphic to the Goel-Okumoto NHPP model and the Musa execution-time model. We can simplify the equation and obtain:

$$m(s) = M(1 - (\frac{1}{K})^{s}),$$

where $K = \exp(\frac{p}{N})$.

Appendix B: Examples of EXMAP Outputs

An example of a report produced by EXMAP is shown in Figure 3.

```
''''''''' EXMAP HIGH LEVEL SUMMARY: PROGRAM AREA DATA                 ''''''''

             DATE:  8/24/92
             TIME: 09:08.42
     TEST CASE ID: TEST4

|<--                      PROGRAM IDENTIFICATION  -->| <--      TEST CASE PERFORMANCE       -->|
|                          |                         | STATEMENTS:          | BRANCHES:         |
| PA LOAD MOD PROC         | LISTING NAME            | TOTAL    EXEC   \\   | CPATH  TAKEN   \\  |
------------------------------------------------------------------------------------------------
  1 TEST4Z   MAIN             TEST4    LISTING          28       20  71.4     26      13  50.0
  2          ADD10                                       2        2 100.0      0       0   0.0
  3          SUB5                                        2        2 100.0      0       0   0.0
------------------------------------------------------------------------------------------------
Summary for all PAs:                                    32       24  75.0     26      13  50.0
 ''''''''' EXMAP HIGH LEVEL SUMMARY: UNEXECUTED CODE                  ''''''''

|<--                      PROGRAM IDENTIFICATION  -->|
|                          |                         |
| PA LOAD MOD PROC         | LISTING NAME            | start    end     start    end     start    end
------------------------------------------------------------------------------------------------------
  1 TEST4Z   MAIN             TEST4    LISTING          20       20        27     27        32     32
                                                        54       55        63     66
------------------------------------------------------------------------------------------------------
 ''''''''' EXMAP HIGH LEVEL SUMMARY: BRANCHES THAT HAVE NOT GONE BOTH WAYS ''''''''

|<--                      PROGRAM IDENTIFICATION  -->|
|                          |                         |
| PA LOAD MOD PROC         | LISTING NAME            |  stmt      stmt      stmt      stmt      stmt
----------------------------------------------------------------------------------------------------
  1 TEST4Z   MAIN             TEST4    LISTING           16        19        22        22        22
                                                         22        29        29        29        29
                                                         45        66
              EXMAP Annotation Symbols

Each  instruction line of the listing has a character to the
right of the statement number to indicate what happened
during the test run:

\\   An unconditional branch instruction that has executed
#   A conditional branch instruction that has executed both ways
>   A conditional branch instruction that has branched but not fallen through
V   A conditional branch instruction that has fallen through but not branched
:   Non-branch instruction that has executed
¬   Instruction that has not executed
```

Figure 3 (Part 1 of 2). Summary of code coverage and annotated C listing

```
 9:         |int main(void) {
10          |        .....
16V      1  |    if (TEST_PARM == yes)           /' this branch never taken        '/
17:      2  |        TEST_VAL = 13;              /' this stmt always executed      '/
19>      3  |    if (TEST_PARM == no)            /' this branch always taken       '/
20¬      4  |        TEST_VAL = 17;              /' this stmt never executed       '/
21          |    /' Note that there is an annotation symbol for each condition    '/
22VVVV      |    if ((TEST_PARM == yes && TEST_VAL == 17) ||
23       5  |        (TEST_PARM == yes && TEST_VAL == 13))
24          |                                    /' this is always true            '/
25:      6  |        TEST_VAL = 5;               /' this stmt always executed      '/
26          |    else
27¬      7  |        TEST_VAL = 10;              /' this stmt never executed       '/
28          |    /' Note that there is an annotation symbol for each condition    '/
29VV>¬      |    if ((TEST_PARM == yes && TEST_VAL == 17) ||
30       8  |        (TEST_PARM == no  && TEST_VAL == 13))
31          |                                    /' this is always false           '/
32¬      9  |        TEST_VAL = 10;              /' this stmt never executed       '/
33          |    else
34:     10  |        TEST_VAL = 5;               /' this stmt always executed      '/
46          |         .........
```

Figure 3 (Part 2 of 2). Summary of code coverage and annotated C listing