Hypothesis Testing for Module Test in Software Development

Tsuneo Yamaura, Hitachi Software Engineering , Yokohama Japan 231-8475, yamaur_t@and.hitachi-sk.co.jp
Akira K. Onoma, Hosei University, Koganei Tokyo 184-8584 Japan, akonoma@k.hosei.ac.jp
Wei-Tek Tsai Arizona State University, Tempe AZ 85287-5406 USA, Wei-Tek.Tsai@asu.edu

Abstract

One of the most important issues in the software development is how to guarantee that the software satisfies the quality defined in the requirement specification. This paper proposes that the issue can be solved, first the number of test cases is statistically calculated from the failure density defined in the requirement specification, then the selected test cases are executed basing on the hypothesis testing.

This paper also presents how our method can be used for debugging. When the number of the test cases is calculated, we applied the statistical behavior of the software quality to the integration testing. We, however, did not consider the ripple effect since it is unable to measure.

In order to guarantee the quality of 4σ *and* 5σ*, we found that many more test cases are needed than is previously believed enough.*

Keywords: hypothesis testing, regression testing, quality model, number of test cases, path expression

1 Introduction

The quality of the module test itself at the early stage of the software development significantly affects the upcoming integration and system test. Based on the quality characteristic of the module test and the ripple effect, it is quite useful to determine the number of the test cases that guarantees the targeting failure density.

This paper explains the solution for these issues and the generation method of the test cases applying the path expression. 2. describes motive and purpose of the main issue. 3. describes the calculation of test cases based on the basic model and integrated module model. 4. examines whether hypothesis testing is applicable to the module test. In this case, it requires the condition that a bug must not be discovered during the test. For this reason, the number of inspection may increase, and the man-power may exceed the expectation. As a means to solve this problem, we proposed that the software quality be evaluated after all the test cases are executed. In 5., the method using a path expression is proposed to prepare a huge number of test cases. Finally 6 presents the conclusion and the future plan.

2 Purpose and Motivation

Software testing detects bugs in the program. If we redesign and execute new test cases, bugs will still be found as before. When all the new test cases are completely executed without bugs, testers believe that there are no bugs[3].

Since software never wears out, a program is always acceptable if the program is implemented as specified, and all the program paths are tested and passed. However, the number of the program paths is almost infinite while the resource for the software development is almost limited, testing all the paths is unrealistic. This paper uses statistical hypothesis testing to guide the testing process when the resource is limited.

This technique has been proven to be effective for the DoD Y2K testing effort [5].

3 The Number of Test Cases

The module testing handles the functions of a single module and integrated modules. If a bug detected in the module testing, the bug must be fixed, and tested again with considering the ripple effect.

We discuss how to determine the number of test cases with considering the basic model, integrated module model, and ripple effect.

3.1 Number of Test cases based on the Basic Model

The module test is considered as completed when the test cases for the target module are developed and executed successfully. The test cases are usually designed to satisfy a certain coverage criteria such as path coverage and condition coverage. However, in practice the developed test cases are not perfect or enough, it is difficult to evaluate the software quality even if all the test cases are successfully executed.

The quality of the module falls within the target reliability and failure density if the reliability and failure density are predefined, the test cases are selected accordingly, and all the test cases pass successfully. The probability that the failure density does not exceed the threshold is determined by the following expression [1].

$$C = 1 - (1 - B)^N \quad (1)$$

where, C is the reliability, B is the targeted failure density of the module, N is the number of the test cases randomly chosen and must be passed without a single failure. (1) is shown in Figure 1.

0730-3157/02 $17.00 © 2002 **IEEE**
IEEE COMPUTER SOCIETY

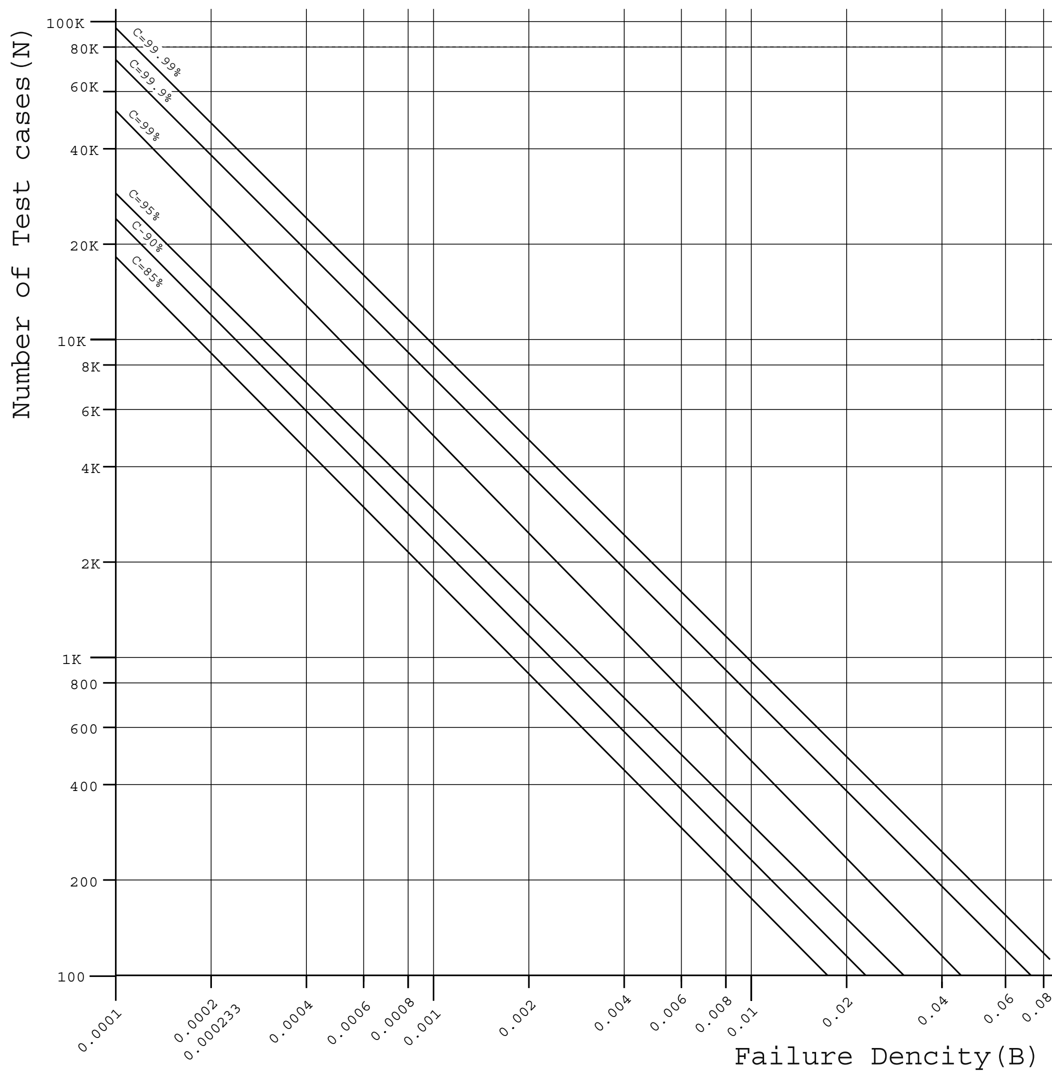

Figure 1: The relation of the failure density and the test case to reliability

The expression (1) shows that when N random test cases are executed without a single failure, one can have the confidence C that the actual failure density of the software is below B. This is a hypothesis testing.

The number N is huge as Figure 1 shows. Even though the path coverage criterion is limited to all the functions or all the possible program paths, the number of the testing paths is still large. Loops make the number of paths easily burst into virtually infinite. Comparing the number of test cases for such path coverage, the number N may look reasonable. Furthermore, the number did not change regardless of the size and complexity of the software.

3.2 Test Case Calculation in the Integrated Module Model

To obtain the expression to calculate the reliability of a module and a subsystem constituting from modules, we extend the original expression to handle more input domains. The premises for the extension are as follows:

(1) Suppose the total number of the input domains is K, and the number of input domains contained in its low rank domains is M. That is, as shown in Figure 2, K includes M.

(2) To obtain the reliability defined beforehand, the number of the test cases which must be performed continuously without failure is assumed N.

(3) Using the reliability C_{K-M} of a higher rank domain and the reliability C_M of a low rank domain M, we assume the reliability C_T of the whole system to be the weighted average of C_{K-M} and C_M whose weights are proportional to the number of the test cases which should be contained in each domain.

IEEE COMPUTER SOCIETY

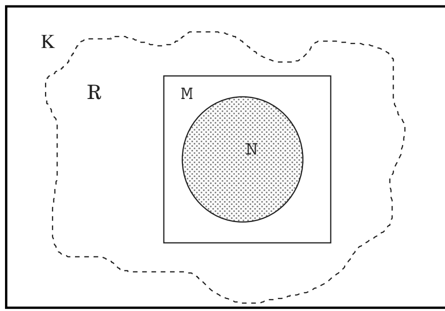

Figure 2: The model with consideration to the integrated test and the ripple effect

Consequently, the reliability C_T of the whole system should be as follows:

$$C_T = (\frac{K-M}{K}) \times C_{K-M} + (\frac{M}{K}) \times C_M \quad (2)$$

Then, the number of the test cases N which must be performed for the low rank domain M is as follows from the expression (1):

$$\begin{aligned} N &= \frac{\ln(1-C_M)}{\ln(1-B_M)} \\ &= \frac{\ln(1-(\frac{K}{M}) \times C_T + (\frac{K-M}{M}) \times C_{K-M})}{\ln(1-B_M)} \end{aligned} \quad (3)$$

Consequently, if the reliability C_T of the whole system is predetermined, and the reliability C_{K-M} of a higher rank domain and the threshold B_M of the failure density are given, the number of the test cases which must be continuously executed to test the specific function in the specific low rank domain is computable by using the expression (3). Refer Figure 3.

When M/K is comparatively small, the number of the mandatory test cases changes drastically, but it does not make a big change if M/K is larger than 0.3 or 0.4. That leads the following:

(1) Even if M/K is comparatively small in the testing of the modified portion, a certain minimum number of the test cases is required to obtain the target failure rate and probability.

(2) When M/K is small, even if M/K changes slightly, the number of the required test cases changes greatly.

3.3 The Number of Test Cases considering the Ripple Effect

New bugs sneak in when source program is modified for the functional change or bug correction. This is a typical example of the ripple effect[5].

The major issue of the ripple effect is that we cannot predict what effect occurs where. Thus we cannot define the number of test cases we have to execute to satisfy the targeted quality. In particular it is extremely difficult to guarantee the correctness of the integrated system which consists from multiple subsystems.

4 Possibility of Hypothesis Module Testing

4.1 Aim of Hypothesis Testing

Although the high quality software definitely needs high quality design, we have to test the quality as the final endorsement. Theoretically it is desirable to test all the possible program paths. However, if we do so, the number of the test cases of a small module easily grows up to 10,000 or 2,000,000 immediately. This is hardly unrealistic.

However the hypothesis testing will be effective on the basis of the following assumptions:

(1) The failure density B and reliability C are defined in the requirement specification.

(2) The quality specified in the requirement specification is assumed to be fulfilled if the test result proves that the failure density is below B with the reliability of C.

(3) If a bug occurs to one of the test cases, the cause should be detected and fixed, and the inspection to catch the same kind bug should be reiterated.

4.2 Test scale in Hypothesis testing

The hypothesis testing reduces the number of the mandatory test cases significantly.

Generally, the scale of the test cases for the hypothesis testing is as follows: we can guarantee the quality with approximately 13,000 test cases (reliability 95%), even if quality of 5σ is required, for example. Five σ means that there are 233 errors in one million operations, and the failure density B is 0.000233. Although 13,000 test cases are not easy to execute, it is not impossible.

When six σ is targeted, first, as a minimum requirement, four σ (failure density 0.00621, 6,210 errors out of one million operations) must be guaranteed, then to perform the β test, for example, of 5,000 to 10,000 persons. For instance, when testing the software of 500KLoc, it is absolutely impossible in the first place to declare that "there is no bugs" with only 3,000 test cases. The probability of hitting no bugs after 1,000 test case execution is at most 95%: this leads an erroneous conclusion once in every twenty trials.

4.3 Determination of Failure density and Reliability

Prior to the testing, determine the upper limit of the failure density and the reliability that the failure density will be below the upper limit. For the very important system like a nuclear reactor, there must be no bug: the reliability is aimed higher and the threshold of the failure density lower. The number of the test cases will increase as the reliability gets higher and the threshold of failure density is lower.

0730-3157/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

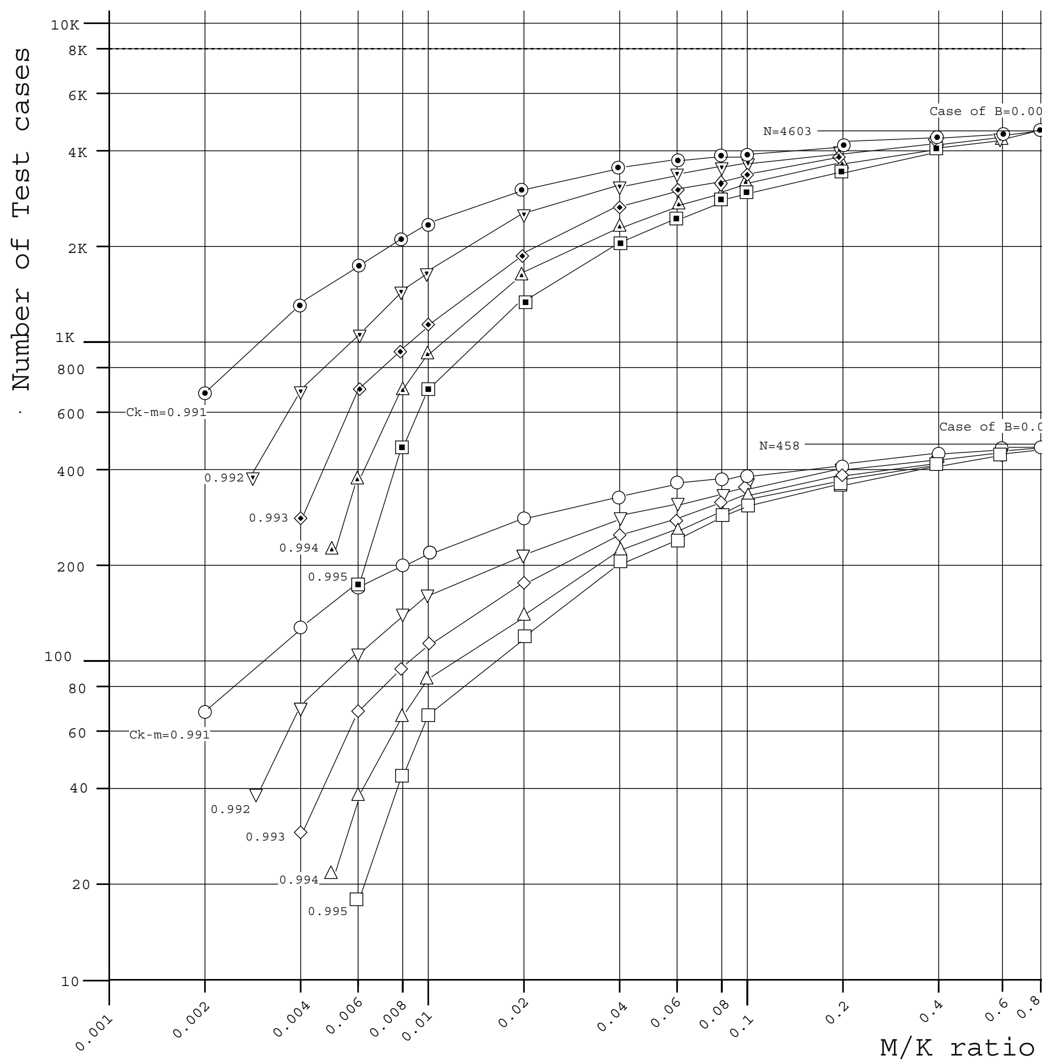

Figure 3: The Number of Test Case when M/K Ratio is changed

We must remind that the failure density is the number of the test cases which can be performed continuously without hitting any failures. That is, the failure density B is the probability which encounters one error in N trials. For example, B is 0.1 if it performs 10 operations with one error, and, similarly, B is 0.01 if one error occurs in 100 trials.

Provided that the failure density B is 0.01, and the 456 test cases are executed without errors. The reliability C of being the failure density B is 0.01, i.e., 99%. That is, when the system is performed 100 times, it is likely to fail only once. For example, N is 687 when B is 0.01 and C is 0.999. If the theoretical number of the test cases needed for a target module is less than N, we can say that the required failure density and reliability are satisfied when all the test cases are executed successfully.

The following estimation can be given if this value is applied to an actual application program:

If an application program is developed for a daily operation for a specific client, i.e., one specific user, one failure will happen in 137 weeks (approximately 2.7 years) in case that the program is executed five times a week. However, if such daily operating application program is developed for arbitrary users, and the number of the users exceeds 687, one failure will happen almost every day. Similarly N is 69,074, when B is 0.001, and N is 6,904 and 0.0001.

4.4 Selection of Test cases

The test case selection must simultaneously consider two characteristics: randomness and independence. From the independent sets of the test cases, you have to randomly choose the test cases whose number is sufficient for realizing the demanded reliability.

When a module has a very few functions, and the number of the selected test cases does not satisfy the number of predetermined test cases, test all of them. If they all pass, it can

IEEE COMPUTER SOCIETY

be considered that failure density is zero.

4.5 Method of Module test

In the module testing, all the modules that are newly developed, have newly added functions, or have functional changes must be tested. The module which does not correspond to such conditions can omit a test.

To confirm that the correction was performed correctly, you have to create new test cases. In this case, the regression test [2] alone is inadequate.

For the module influenced by the added or modified functions, and the module that has a data exchange with such module, you have to confirm that the added or changed function correctly handles the inputs and outputs.

It is presumed that the reliability demanded has realized if a test is passed. When that is not satisfied, you have to test once again.

You have to carry out a certain coverage test. A key point of the success for a high quality system is that the path coverage criteria should be stricter. C_0 nor C_1 is not sufficient but the program path complete coverage is desirable. However, the path coverage by viewing, not by the machine execution, is also allowed in some cases.

4.6 Reliability Evaluation in case Bugs are extracted

The argument so far had the premise that a bug is not pointed out in any test cases. That is, this system is applicable only when all test cases pass in the software testing.

It is not rare that the test has to be done once again by another set of test cases because there is only one bug detected. In such case, the quality of module(s) should be calculated theoretically, and if the demanded quality is satisfied, it may be possible to apply a special procedure that the concerned bug is fixed later.

It is convenient if the reliability can be estimated when failures exist.

4.6.1 Re-definition of reliability

The purpose here is to be able to measure the reliability of the assumption that the failure density is below B when bugs are detected in the testing. If such reliability is better than the pre-determined acceptable reliability, we can assume that the software passes the test. For this reason, we have to define an objective reliability standard.

Suppose that N test cases are selected randomly, and tested, and M test cases hit bugs. If the actual failure density is more than B, the probability that the number of the detected bugs is below M when N test cases are executed is below F. F shall be defined by the following expression:

$$F = \sum_{k=1}^{M} \binom{N}{k} B^k (1-B)^{N-k}$$

In other words, when the failure density of the software is more than B, the probability that N test cases reveal more than M bugs is at least $1-F$. Consequently, as the probability that more than M bugs will be detected, the reliability C will be quantifiable as follows:

$$C = 1 - F = 1 - \sum_{k=1}^{M} \binom{N}{k} B^k (1-B)^{N-k} \quad (4)$$

$$= 1 - \sum_{k=M+1}^{N} \binom{N}{k} B^k (1-B)^{N-k} \quad (5)$$

Intuitively as the probability of detecting more bugs than in the actual testing grows higher, the reliability that the failure density does not exceed the upper limit becomes higher.

The probability that the conclusion, the failure density is lower than B, is not correct is $1 - C$.

4.6.2 How to decide reliability

We assumed that the reliability C was the probability that the targeted failure density has an upper limit, B. Consequently, in the actual software development, each project sets the targeted reliability with applying the same idea as to decide the reliability stated by 4.3.

4.6.3 Example of failure density

When the threshold of the failure density is fixed to $B = 0.05$, and the number of the test cases is changed, Figure 4 shows how the reliability changes with the number of the bugs detected during a test.

To acquire the reliability of 0.95 for the failure density threshold of $B = 0.05$, the number of the bugs allowed in 100 test cases should be only one.

If the target reliability is 0.8, it is obvious that the number of the bugs in 100 test cases must be less than two. Note that the number of the acceptable bugs does not grow drastically even if the reliability drops. The reliability falls abruptly as the number of the bugs increases. This tendency becomes more obvious when the threshold of the failure density is smaller.

5 Tools for Test Case Creation

It is problematic to substitute a module test only for whitebox test. It is required to select the test case selected from the both view points of whitebox testing and blackbox testing, and to review the test case with the requirement specifications and software design specifications.

The test case for a module test can be created comparatively easily if a tool, that mechanically handles path expressions, is used to extract test cases from a source program. The points for the test case extraction are as follows.

IEEE COMPUTER SOCIETY

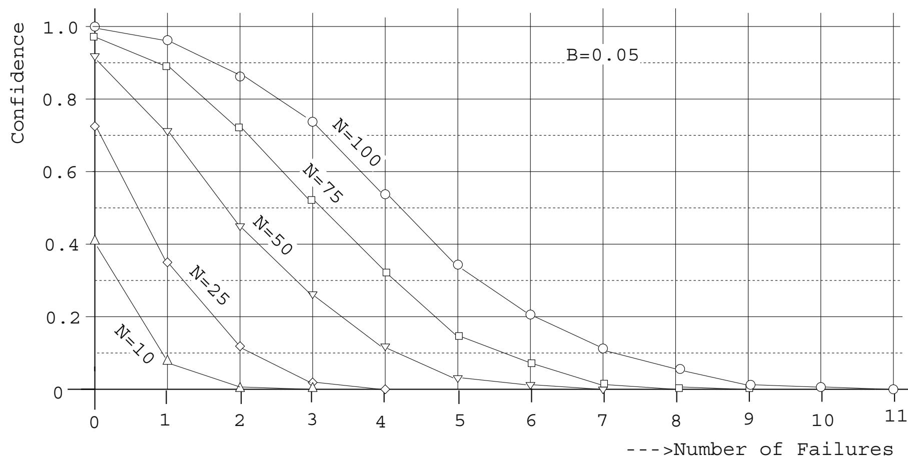

Figure 4: Change of the reliability by the increase in a bug

(1) Decompose the path expression obtained from the source program into the iterated part and the non-iterated part.

(2) The portion which does not contain the repetition part in a path expression is one function by itself.

(3) Divide the repetition part into no loop, one loop, two or more loops, and determine each input domain. Connect each part with the non-iterated part, match the connected parts with a function in the requirement specification, and make it one test case. Note that this test case changes as the input data domain changes. Also apply those operations to the design specification.

(4) The number of iteration depends on a program. Generally a repetition part needs to consider five test cases of 0-, 1-, $(n-1)$-, n-, and $(n+1)$-time execution (n is a maximum number of repetition).

6 Conclusion

Although the man-power of a module test is needed more than in the conventional testing, the bug extracted at the integrated test phase is decreased, in particular the interface bug is decreased. That is, the test phase at which a bug is extracted became earlier. This is the effect of our test method.

The following things can be concluded by this technique:

(1) The quality estimation is easier than the method of predicting the number of bugs using Gompertz curve [4]. It also requires less person-month but needs highly skilled decision.

(2) The test cases for the module test can be completely expressed in terms of the path expression of the concerned module, and a mere repetition can be deleted.

(3) The code reviewing can be performed in a more thorough fashion, and a program can be developed in a manner that a module needs less test cases, and the programming that requires less test cases.

(4) The number of the mandatory test cases increases compared to the former method. However if a test case can be semi-automatically created from the path expression extracted from the source program, the person-power for the test case generation will not increase so much. Other test case creation tools, such as Capture/Replay, are also indispensable.

(5) To be able to extract test cases in a random manner, a special consideration is needed to classify the test cases.

As a secondary effect, the reusability of a module becomes much higher because the module is thoroughly tested by this technique. We completed the evaluation of our new test method with an experiment in a small project.

We have been improving our method with implementing the semi-automatic test case generation. Also we are on the process of analyzing the magnitude of the ripple effect.

References

[1] W. H. Howden. Good Enough versus High Assurance Software Testing and Analysis Methods. In *Proc. of IEEE HASE*, pages 166–175, 1998.

[2] A. K. Onoma, W. T. Tsai, H. Mustafa, M. Poonawala, and H. Suganuma. Regression Testing in an Insdustrial Environment. *Communications of the ACM*, pages 81–86, May 1995.

[3] A. K. Onoma, Tsai W. T, F. Tsunoda, H. Suganuma, and S. Subramani. Software Maintenance– Industrial Experience. *Journal of Software Maintenance*, 7(2):333–375, December 1995.

[4] Akira K. Onoma and Tsuneo Yamaura. Practical steps toward quality development. *IEEE Software*, pages 68–77, September 1995.

[5] W. T. Tsai, R. Paul, W. Shao, S. Rayadurgam, and J. Li. Assurance-based Y2K Testing. Private communication, Software Engineering Laboratory, Department of Computer Science and Engineering, University of Minnesota, December 1999.

IEEE COMPUTER SOCIETY