Chapter

13

Software Testing and Reliability

Joseph R. Horgan
Bellcore

Aditya P. Mathur
Purdue University

13.1 Introduction

It is believed that there is an important relationship between the estimation of reliability of a program, its structure, and the amount of testing it has been subjected to. Though one can imagine several ways of quantifying the amount of testing, we consider one or more measures of code coverage as possible quantifiers. Statement coverage, decision coverage, and data flow coverage are some of the code coverage measures. These measures are based on the structure, often detailed, of the software. Several software reliability theorists observe that the structure of the software should be closely followed in the analysis of reliability. [Broc90] suggests: "At some future time it may be possible to match a reliability model to a program via the characteristics of that program, or even of the software development methodology used."

[Musa87] has suggested a similar possibility. The importance of distributing testing according to a user's operational environment is a central theme of reliability estimation. In this context, characterization in [Goel85] is particularly apt: "To illustrate this view of software reliability, suppose that a user executes a software product several times according to its usage profile and finds that the results are acceptable 95 percent of the time. Then the software is said to be 95 percent reliable for that user."

As per the above quote, a program is tested according to its usage profile and then one or more of several reliability models applied to the failure data to obtain reliability estimates. Such an approach to reliability estimation fails to account for the difficulties in assessing accu-

rate usage profiles and in accounting for the structure of the software. In this chapter we point out some of these difficulties and suggest two approaches for reliability estimation. Our approaches make use of code coverage explicitly in the estimation process, whereas the existing time-domain approaches as outlined in [Musa87] do not.

The remainder of this chapter is organized as follows. In Sec. 13.2 we present some concepts related to software testing. The problems encountered in establishing an operational profile are addressed in Sec. 13.3. The effect of nonavailability of an accurate operational profile on reliability estimation is also discussed in this section. Our first approach that combines the existing time-domain approaches of reliability estimation with coverage information obtained during system test is presented in Sec. 13.4. Another approach to viewing software reliability is to consider the risk associated with a program. In Sec. 13.5.1 we outline a model to estimate the risk associated with a program. This model does not use the time-domain approach in any form. Instead it uses various measures of code coverage to estimate risk.

13.2 Overview of Software Testing

A key question we address in this chapter is: "How does the nature of testing affect reliability estimation?" Below we lay the groundwork for an answer to this question by examining white-box testing.

13.2.1 Kinds of software testing

There are many ways of testing software. The terms *functional, regression, integration, product, unit, coverage,* and *user-oriented* are only a few of the characterizations we encounter. These terms are derived from the method of software testing or the development phase during which the software is tested. The testing methods *functional, coverage,* and *user-oriented* indicate, respectively, that the functionality, the structure, and the user view of the software are to be tested. Any of these methods might be applied during the *unit, integration, product,* or *regression* phases of the software's development. In this context the unit phase is the coding of small software components, the integration phase puts units together into larger components, and the product phase integrates the software into its final form. Regression testing pertains to the re-release of a modified software product.

13.2.2 Concepts from white-box and black-box testing

White-box, or coverage, testing uses the structure of the software to measure the quality of testing. It is this structural coverage and its measurement that we believe is of value in reliability estimation. We describe two coverage testing methods: mutation testing and data and

control flow testing. Subsequently, we discuss the use of these methods in reliability estimation.

1. *Statement coverage* testing directs the tester to construct test cases such that each statement or a basic block of code is executed at least once.

2. *Decision coverage* testing directs the tester to construct test cases such that each decision in the program is covered at least once. A decision refers to a simple condition. Thus, for example, the C language statement **if** $(a < b \,||\, p > q) \ldots$ consists of two simple conditions, $a < b$ and $p > q$, and one compound condition. We say that a decision is *covered* if during some execution it evaluates to true and in the same or another execution it evaluates to false. In the above example, the two simple conditions must evaluate to true and false during some execution for the decision coverage criterion to be satisfied.

3. *Data flow coverage* testing directs the tester to construct test cases such that all the def-use pairs are covered. Consider a statement $S_1 : x = f()$ in program P, where f is an arbitrary function. Let there be another statement $S_2 : p = g(x,*)$ in P, where g is an arbitrary function of x and any other program variables. We say that S_1 is a definition and S_2 a use of the variable x. The two occurrences of x constitute a def-use pair. If the use of a variable appears in a computational expression, then such a pair is termed as a c-use. If the use appears inside a predicate then the pair is termed as a p-use. A path from S_1 to S_2 is said to be *definition-free* if no statement along this path, other than S_1 and S_2, defines x. Such a path is considered feasible if there exists at least one $d \in$ the input domain D such that when P is executed on d the path is traversed.

All statements in P that can possibly be executed immediately after the execution of some statement S are known as *successors* of S. We say that a c-use or a p-use is *covered* if the execution of P on some set of test cases causes at least one definition-free path to be executed from the defining statement to the statement in which the use occurs and to each of its successors. Further details of data flow testing may be found in [Horg90].

4. *Mutation testing* helps a tester design test cases based on a notion very different from that of path-oriented testing strategies such as the ones described above. Given a program P, mutation testing generates several syntactically correct *mutants* of P. A mutant is generated by making a change in P in accordance with a predefined set of rules. For example, one mutant, say M, of P can be generated by removing a statement from P. We say that a test case d distinguishes M from P if $P(d) \neq M(d)$. M is considered equivalent to P if $\forall d \in D, P(d) = M(d)$. Mutation testing requires a tester to generate test data that distinguish all nonequivalent mutants of P. Further details of mutation testing may be found in [Chio89].

Note that each of the four testing methods provides an *adequacy criterion* against which a test set can be evaluated. We say that a test set T consisting of one or more test cases is adequate with respect to the decision coverage criterion if all the decisions in the program are covered when executed against elements of T. T is adequate with respect to to the p-use criterion if all p-uses have been covered by T.* T is considered adequate with respect to mutation criteria if it distinguishes all the nonequivalent mutants of the program. Each of the preceding adequacy criteria is precise and measurable. Note that functional testing does not provide any such precise and measurable criteria.

It can be shown formally that if a test set is p-use or c-use adequate, then it is also decision adequate [Clar89]. We therefore say that data flow coverage *subsumes* decision coverage. Similar relationships have been investigated empirically among functional, data flow, and mutation testing. Evidence available so far [Math91, Wong93] suggests that test data which are mutation adequate are likely to be data flow adequate whereas a data flow adequate test set is less likely to be mutation adequate. Empirical evidence presented in Sect. 13.4.6 also suggests that even after a significant effort has been spent in functional testing, the test data so developed are not data flow adequate, and hence not mutation adequate. On the contrary, it has been shown [Howd80] that for several types of errors, structural testing is not sufficient, but functional testing is. Further, functional testing appears to be a necessity in any testing activity as it is the first step to verifying that the specific functions that a program is supposed to perform are indeed performed correctly.

Thus, taking empirical evidence, theoretical hierarchy [Clar89], and the practice of software testing into account, we justify the assumption that testing is carried out using a commonsense method first, i.e., functional testing followed by structure-based methods in the order of their expected cost benefits. Despite the existence of various testing methods, it is of interest to note that according to the current industrial practice, only functional testing is carried out in most environments.

13.3 Operational Profiles

A widely accepted definition of software reliability (R) is that it is the probability of failure-free operation of a system. By *system* we refer to the program (P) whose reliability is to be estimated. A variety of models have been proposed for estimating software reliability. Most of these are based on probabilistic principles. The ones that are popular

* There are several other data flow criteria that we have not mentioned in this paper. Definitions of the well-known data flow criteria are provided in [Clar89].

among researchers and practitioners make use of software failure data to estimate R. In reliability growth modeling, the failure data are obtained by testing P on a stream of inputs also known as test cases. Each test case (d) is a point in the input domain of P. To generate a test case the input domain is sampled based on an *operational profile.* As defined in Chap. 5, an operational profile is a list of occurrence probabilities of each element in the input domain of P. When P fails during testing, the time of failure is recorded and the software repaired. This process continues until some form of convergence of reliability estimates is achieved.

For most software, the input domain is extremely large and may be considered infinite for practical purposes. Thus, determination of the operational profile becomes a task to reckon with. Nevertheless, the operational profile is key to reliability estimation for most existing models used in practice, e.g., the Musa-Okumoto model. Not surprisingly, a significant amount of work has gone into developing procedures for estimating an operational profile. Chapter 5 also provides a detailed methodology of how an operational profile can be built. This methodology is dependent on input from the customer. In fact the first step in this methodology is to identify a customer profile. This profile is refined in a sequence of steps to obtain the operational profile, which in turn is dependent on how the users are expected to use the software.

Once an operational profile has been determined, you need to sample test cases from the input domain as per the occurrence probabilities. The software is then exercised against these test cases to obtain failure data. These data are then input to one or more models to estimate reliability.

In the procedure outlined above, the software is treated as a black box. Test cases are generated to test for specific *features* of the software. It has been suggested in [Musa93] that the number of test cases should be limited to several hundred or several thousand. However, neither the test case development nor the reliability estimates have any explicit relationship with how the software is exercised during testing; indeed, an implicit relationship exists. It is this lack of an explicit relationship that forms the basis of our argument against a purely black-box approach to reliability estimation.

13.3.1 Difficulties in estimating the operational profile

As mentioned earlier, an operational profile is an estimate of the relative frequency of use of various inputs to the program. The frequency is user dependent and is necessarily derived by some sort of usage analysis. Below we identify situations under which deriving an accurate

operational profile may not be possible and hence you may have to rely on educated guesses. From our personal experience in software development and discussions with various development groups, we have come to believe that each of the scenarios described below is encountered by one or more developers and is not fictional. We also point out problems that may arise when an operational profile is inaccurate. The problems cited below are often encountered and discussed informally by practitioners; we are not aware of documentation.

New software. When a new system is designed, as opposed to modifying an existing system, one may not have any customer base for this system. As an example, consider the development of a system that will control an instrument for experimentation aboard a spacecraft. The experiment is one of its kind and has never been performed before. Features in this system will correspond to the requirements derived from an analysis of the instrument and the nature of its expected use. It is therefore likely to have a list of features but no existing customer base. Thus, we have to necessarily rely on guesstimates of occurrence probabilities for various features. If the system is designed to be fault tolerant, then one needs to guess the probabilities of failure of the application modules in the system. These probabilities will in turn determine the probabilities of how certain features of the fault manager will be exercised. Such failure probabilities may depend on a variety of relatively well understood phenomena (such as hardware failure) and not so well understood phenomena (such as data corruption due to cosmic rays). This is likely to add an extra degree of uncertainty to the occurrence probability estimates of features of the fault manager.

New features. New versions of an existing system may be continually under development. A new feature is added to the system assuming that one or more users will use it. Even though there exists a user base for the existing version of the system, there is no user base for the new version yet to be released. Once again the developer has to rely on guesstimates of the occurrence probabilities of the new features. The problem is further complicated by new features that might significantly alter the usage pattern of the existing system. If the users like a new feature they may not use an existing feature, thereby altering the usage frequency of this existing feature. Such a change may be difficult to anticipate, and the guesstimates of occurrence probabilities of various features could be very inaccurate.

Feature definition. A feature is often not a well-defined entity. For example, suppose that a system provides two features f_1 and f_2. Then, is the use of these features in different sequences also a feature? Adding

all possible sequences of features into the operational profile might result in a very large profile, which is difficult to build and manage. As another example, suppose that a sort module is embedded in a large system. The module is internal and the user cannot access it directly. However, it supports two features f_1 and f_2. The system provides a feature f that occasionally uses the sort module. Should the operational profile treat only f as a feature or the combinations f,f_1, and f,f_2 as two distinct composite features?

Feature granularity. Consider a program that has a total of N lines of executable code. If the operational profile consists of k features, then the program has an average of k/N lines per feature. In practice there may be more or less lines per feature than the average. For a system with, for example, 100K lines of code and 100 features in the operational profile, we are likely to find features that correspond to more than 1000 lines of code. In order to test the code well, it would be desirable to specify the features with finer granularity, resulting in lesser lines of code per feature. However, if features are based on what a user uses directly, it may not be possible to specify features with a fine granularity.

Multiple and unknown user groups. An operational profile is intended to model one or more users. It is assumed that these users belong to a relatively homogeneous class. A reliability estimate given under such an operational profile is at best valid for the class of users for which the profile has been developed. A developer, such as the one who develops an operating system, might prefer to provide reliability estimates for the system independent of the user. It is not clear how to estimate an operational profile to meet such a requirement.

The above discussion leads us to believe that estimation of operational profiles is a difficult and error-prone task. Reliability estimation using operational profiles is often projected as "customer or user-centered" testing of a system. Based on arguments given above, we believe that it is at best a "known user-centered" approach. If an unknown user uses the system in a way that does not match with the operational profile, then there is no guarantee that the projected reliability figures will hold.

13.3.2 Estimating reliability with inaccurate operational profiles

As mentioned above, existing and popular models of reliability estimation make use of black-box testing to generate failure data. Below we argue that this approach might yield failure data leading to optimistic reliability estimates. Experimental evidence in support of our arguments appears in [Chen94b].

Inadequate test set. When using black-box testing based on an operational profile, the input tests are based on the features in the profile. A feature simply may not appear in an erroneous profile or the probability of its occurrence may have been erroneously estimated to be too low. This could result in not testing a feature or testing it scantily. The problem with such testing is that there is no program-based notion of the adequacy of a test set. Thus, after a program has been tested and you obtain the failure data, you have no way of determining how "good" the set of test cases was. The notion of adequacy that is used in such testing rests upon the reliability of statistical sampling from the input domain using the operational profile. This notion does not account for the fact that an inaccurate profile might result in a poor test set.

Thus, for example, if a feature is not tested at all, or tested but not thoroughly, and if there are errors in the implementation of that feature, then the failure data may not contain failures resulting from these errors. Such failure data might lead to an overestimate of reliability.

Coarse features. A feature could correspond to several lines of code. While using black-box testing, we construct several test cases to exercise the feature thoroughly. However, there is no measure of how well the feature has been exercised. There might be parts of the code related to this feature that never get exercised even though the feature occurs with a high probability in the operational profile. This is likely to happen when test cases are being randomly sampled from the input domain or a tester is generating them manually without a knowledge of how well the code corresponding to this feature has been exercised so far. Empirical data obtained from two applications that had been tested extensively over several years have indicated that indeed tests generated manually using a knowledge of program features and the functions used to implement them is insufficient to obtain a high level of code coverage [Horg92].

Inadequate testing of a feature is likely to result in misleading failure data and inaccurate reliability estimates. Note that we are assuming an accurate operational profile in this case. We have pointed out that despite this accuracy there is a possibility of obtaining misleading failure data.

Interacting features. In large systems, features often interact in a variety of ways [Zave93]. A simple form of interaction occurs when, say, feature f_1 works correctly when exercised before exercising feature f_2, but not otherwise. In large systems we deal with several hundred features. It is not clear how such interaction can be checked systematically when performing black-box testing. Once again, failure to check for faulty interactions may generate misleading failure data, leading to inaccurate reliability estimates.

Above we have listed several reasons why reliability estimates may be inaccurate. Currently there is a lack of data suggesting the extent of this inaccuracy. We are aware of one study which provides evidence in favor of the conjecture that an inaccurate operational profile may have a serious effect on reliability estimates [Chen94b]. Below we describe an approach that incorporates knowledge gained during white-box testing into reliability estimation, with the aim of reducing the effect of operational profile errors on reliability estimates.

13.4 Time/Structure-Based Software Reliability Estimation

An estimate of the probability of software failure within a specified time of operation is an important and useful metric. Such a metric, referred to as *software reliability,* is useful to both the software developer and its user. The developer can use this metric to decide whether to release the software or not. The user can decide whether to begin using the software or not at a given time. The importance of software reliability was realized several years ago and has been a major subject of research in software engineering. A large number and variety of models have been proposed to estimate software reliability. Often, these models have also been applied to data obtained from working software [Musa87]. The accuracy of these models, as measured against the predicted versus actual software failure, has varied from one project to another.

In the remainder of this chapter we describe two methods for estimating software reliability and the underlying rationale. A key feature of our methods is that they account for the fine structure of the software under development. This feature distinguishes our methods from the existing methods that employ time-domain models; it is also the basis of our claim that structure-based reliability methods are likely to provide more accurate reliability estimates than the existing time-domain-based methods.

13.4.1 Definitions and terminology

Let P denote a program under test whose reliability is to be estimated. During testing, P is executed on a test case d selected from the input domain D. The output of P obtained by executing it on d is denoted by $P(d)$. Each execution of P requires a test case and some CPU time. We assume that *testing effort* is measured either in terms of the number of test cases on which P is executed or through the cumulative CPU time for which P has been executed. The CPU time spent in executing P during testing is used in time-domain reliability models based on execu-

tion time. Let T_k denote the time at which the kth failure occurs and N_k the number of test cases used by time T_k. We define E_k, the effort spent in testing, as follows:

$$E_k \triangleq \begin{cases} T_k - T_{k-1} & \text{for time-based model} \\ N_k - N_{k-1} & \text{for test-case-based model} \end{cases} \tag{13.1}$$

Let e_i denote the effort spent during the ith execution of P. Then E_k can be expressed as

$$E_k = \sum_{i=l_1}^{l_2} e_i \tag{13.2}$$

where e_{l1} and e_{l2}, respectively, denote the effort spent in the first and last executions of P during the kth failure interval.

The reliability R of P is defined as the probability of no failure over the entire input domain. More formally, we have

$$R = \mathrm{P}\{P(d) \text{ is correct for any } d \in D\} \tag{13.3}$$

This definition can be cast in terms of a time-based definition. Let the cumulative effort S_k be defined as

$$S_k = \sum_{i=1}^{k} E_i \tag{13.4}$$

In the literature [Goel85], *reliability* is defined as the probability that a software system will not fail during the next x time units during operation in a specified environment. Here x is known as the *exposure period.* A more precise definition is given in [Yama85] using E_k and S_k as follows:

$$R(x \mid t) \equiv \mathrm{P}\{E_k > x \mid S_{k-1} = t\} \tag{13.5}$$

where $R(x \mid t)$ denotes the reliability during the next failure interval of x units, given the failure history during t units. $R(x \mid t) \to R$ as $x \to \infty$ if the test inputs are operationally significant.

13.4.2 Basic assumptions

Most models rely on certain assumptions that are often not satisfied in practice. Chapter 3 has identified these assumptions for various models. Several researchers have examined the validity of these assump-

tions. Here we examine one fundamental assumption: namely, the assumption that testing is carried out in accordance with the operational profile. This implies that the testers know and make use of the operational profile of the inputs. A knowledge of the operational profile implies knowing the frequency distribution with which test inputs are expected to be encountered when the software operates in its intended environment (Chap. 5). This frequency may presumably be used to decide which test inputs must be used during testing and in what order. Obviously, test cases not encountered during the monitoring of the environment will correspond to a frequency of zero in the profile. In Sec. 13.3 we have examined situations where an accurate operational profile may not be available.

The reliability models proposed herein impose a testing methodology on the tester. Such a methodology ensures (1) improved data input to a reliability model and a better reliability estimate and (2) that the predictions are less sensitive to the possible differences between the true operational profile and its approximation derived during testing.

13.4.3 Testing methods and saturation effect

We begin by describing a *saturation effect* that is associated with all testing methods. An understanding of this saturation effect is a key to the realization of the shortcomings of the application of existing models. A saturation effect refers to the tendency of an individual testing method to attain a limit in its ability to reveal faults in a given program. Figure 13.1 illustrates the saturation effect. As explained below, it is this limit that may cause significant over- or underestimates of reliability using existing models. The method presented herein accounts for the saturation effect [Chen92b].

13.4.4 Testing effort

As testing progresses, data (including calendar time, CPU time spent in executing the software under test, and the number of test cases developed) become increasingly available. Here we refer to the *later* phases of testing (e.g., the system test phase). A reliability model often uses some of the failure data generated during this phase. Musa's basic execution time model (see Sec. 3.3.4), for example uses the total CPU time spent executing the program under test; other researchers have used the number of test cases [Cheu80]. In our discussion below, we consider the CPU time and the number of test cases as indicators of *testing effort.* Thus, as testing effort increases, faults are discovered and removed. This results in an increase in program reliability. We shall denote the testing effort by t_x, where x indicates the testing method that was in use when the effort was measured.

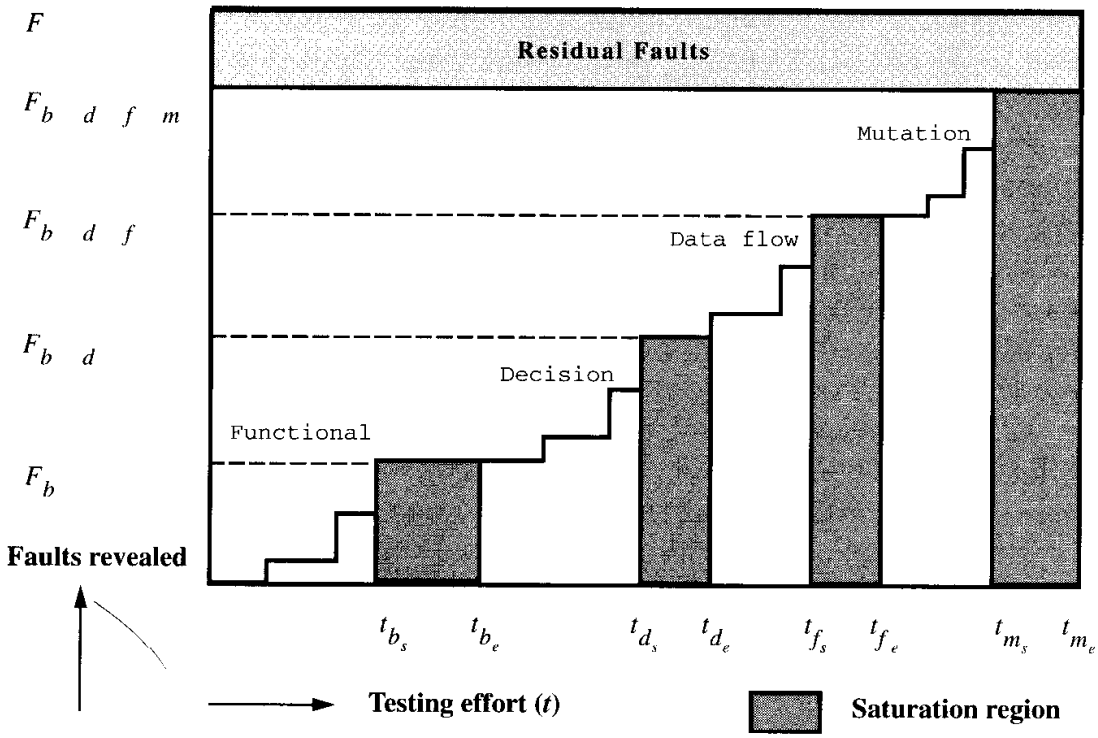


Figure 13.1 Saturation effect of testing methods assuming that the testing methods are applied in the following sequence: functional, decision, data flow, and mutation.

13.4.5 Limits of testing methods

At the start of testing, a program contains a certain number, say F, of faults. As testing proceeds the number of remaining faults decreases. However, when applied, each testing method has a limit on the number of faults that it can reveal for a given program. As shown in Fig. 13.1, we assume that for functional testing this limit is reached after t_{bs} units of testing effort has been expended. Also, functional testing has revealed F_b out of F faults when its limit is reached. In general, $F \geq F_b \geq 0$. Due to its imprecise nature, it is difficult, if not impossible, to determine when functional testing has been completed. In practice, a variety of criteria, both formal (e.g., a reliability estimate) and informal (e.g., market pressure), are applied to terminate functional testing. If no other form of testing is used, this also terminates testing of the product. Note that [Dala90] has formulated a method to decide when to stop testing.

It is reasonable to assume that as functional testing proceeds, the reliability of the software being tested grows when faults found are removed. However, once its limit has been reached, no additional faults are found. A tester, not knowing that the limit is reached, continues testing without discovering any more faults. If existing models for reliability estimation are used, e.g., the NHPP model of Goel and Okumoto [Goel79] then as functional testing proceeds beyond its limit the com-

puted reliability estimate improves even though the reliability of the program remains fixed. The reliability estimate can be improved to any arbitrary limit by increasing the number of test cases executed in the saturation region.

In accordance with the scenario charted earlier, let us suppose that after t_{be} units of functional testing, we switch to decision-coverage-based testing. New test cases are developed to cover the yet uncovered decisions. Eventually, the limit of decision coverage is reached after a total of t_{ds} units of testing. At this point $F_{b\cup d}$ faults have been revealed, with F_d being the number of faults revealed by decision coverage. Note that at this point 100 percent decision coverage may not have been achieved. However, the tester does not know that the limit has been reached and continues testing until t_{de}, by which time all decisions have been covered. Once again, empirical evidence suggests that $0 \leq F_b \leq F_{b\cup d} \leq F$.

We assume that the next switching occurs at time t_{de} to data flow testing and then at time t_{fe} to mutation testing. As shown in Fig. 13.1, the limits of data flow and mutation are reached at times t_{fs} and t_{ms}, respectively, with a total of $F_{b\cup d\cup f}$ and $F_{b\cup d\cup f\cup m}$ faults revealed. We also assume that in general $0 \leq F_b \leq F_{b\cup d} \leq F_{b\cup d\cup f} \leq F_{b\cup d\cup f\cup m} \leq F$.

13.4.6 Empirical basis of the saturation effect

It is possible to construct examples of programs to show theoretically that every structure-based testing method will eventually reach its limit and thus fail to reveal at least one or more faults. This saturation effect has been illustrated for several structure-based methods by a few empirical studies in the past [Budd80, Girg86, Wals85]. Due to its imprecise nature, a proof that saturation effect holds for functional testing does not appear to be feasible, although Howden's work [Howd80] does provide some empirical justification. Below we present empirical justification based on another study with two relatively larger programs than the ones considered by Howden.

TEX [Knut86] is a widely used program in the public domain. It has been tested thoroughly for several years by Knuth [Knut89] and, as a result, a widely distributed test set [Knut84], named TRIPTEST, is available for testing TEX. Prior to installation, it is recommended that TRIPTEST be used to ensure that TEX indeed functions as intended by its author. TRIPTEST has been devised by Knuth primarily to test the functionality of TEX. As indicated in [Knut84], TRIPTEST exercises TEX in several ways that may be highly improbable in practice. Knuth has also documented [Knut89] all the errors discovered during the debugging and use of TEX. An examination [Demi91] of this list of over

850 errors of various kinds indicates that in spite of a fiendish amount of functional testing, errors have persisted in TEX. The above observation is indeed true for yet another UNIX® utility, namely, AWK, which has been tested for several years by Kernighan based on its functionality. However, errors continue to crop up, though with decreasing frequency, in AWK.

We used TEX and AWK to determine how much structural coverage is obtained using test data that has been derived from several years of functional testing. Using a data flow testing tool named ATAC [Horg94, Lyu94b], we decided to compute various coverage measures when TEX, Version 3.0, is executed on TRIPTEST. Table 13.1 lists these coverages. Notice that none of the four structural coverages is 100 percent. It is indeed possible that many of the blocks, decisions, p- and c-uses are indeed either infeasible or can be executed only under rare run-time conditions. Knuth does mention the fact that some parts of TEX that are related to such error conditions are not exercised by TRIPTEST [Knut84].

To ensure that not all of the uncovered structural elements of TEX correspond to error conditions, we examined the uncovered blocks and decisions and identified a few that are not related to processing error conditions arising at run time. Three such blocks, selected arbitrarily, were then removed from the original TEX code and TEX rebuilt. The rebuilt version was then executed against the TRIPTEST. The output generated by the rebuilt TEX was identical to that of the original TEX, thus showing that indeed TRIPTEST did not exercise the removed blocks.

An analysis similar to the one described above for TEX was also carried out for AWK. Several uncovered, though feasible, structural elements were discovered. These analyses strongly suggest that (1) intensive functional testing may fail to test a significant part of the code and, therefore, (2) may fail to reveal faults in the untested parts of the program. It is this empirical observation that justifies our claim that the saturation effect is exhibited by functional testing and that coverage data must be used during reliability estimation.

TABLE 13.1 Coverage Statistics of TEX and AWK

	Coverages (%)			
Program	Block	Decision	p-use	c-use
TEX	85	72	53	48
AWK	70	59	48	55

13.4.7 Reliability overestimation due to saturation

We now argue that the saturation effect can lead to overestimation of the reliability. Figure 13.2 shows the reliability R as faults are removed, and the estimate of R denoted by $\overline{R}$. The testing effort axis is labeled the same as in Fig. 13.1. Assuming that faults are independent, R increases as faults are removed from the program. $\overline{R}$ may, however, increase or decrease as faults are removed. This nonmonotonic behavior of $\overline{R}$ is due to the time dependence of the input data used by most models. Thus, for example, increasing interfailure times will usually lead to increasing $\overline{R}$ obtained from the Musa basic model.

We assume that the $\overline{R}$ generated by a model is a stochastically increasing estimate. This implies that even though $\overline{R}$ may fluctuate, it will eventually increase if the number of remaining faults decreases.

In Fig. 13.2, we indicate that as functional testing progresses, and faults are discovered and corrected, $\overline{R}$ increases. In general, however, it is not possible to detect when the saturation point, t_{bs} in this case, has been reached. Thus, testing may continue well past the saturation point. As shown in the figure, testing in the saturation region does not increase R, though it does increase $\overline{R}$. The increase in $\overline{R}$ is explained by observing that the last value in the interfailure data, i.e., $(t - t_{bs})$, is increasing without any new fault being detected. This increase in $\overline{R}$, while R remains constant, can lead to a significant overestimate of the

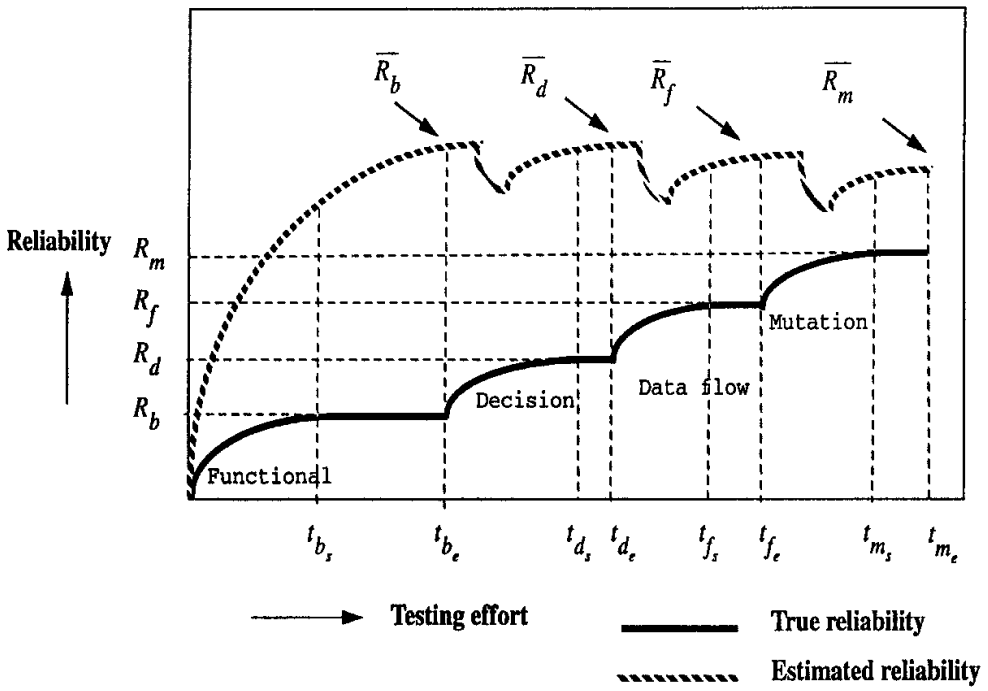


Figure 13.2 Overestimation of reliability due to saturation effect.

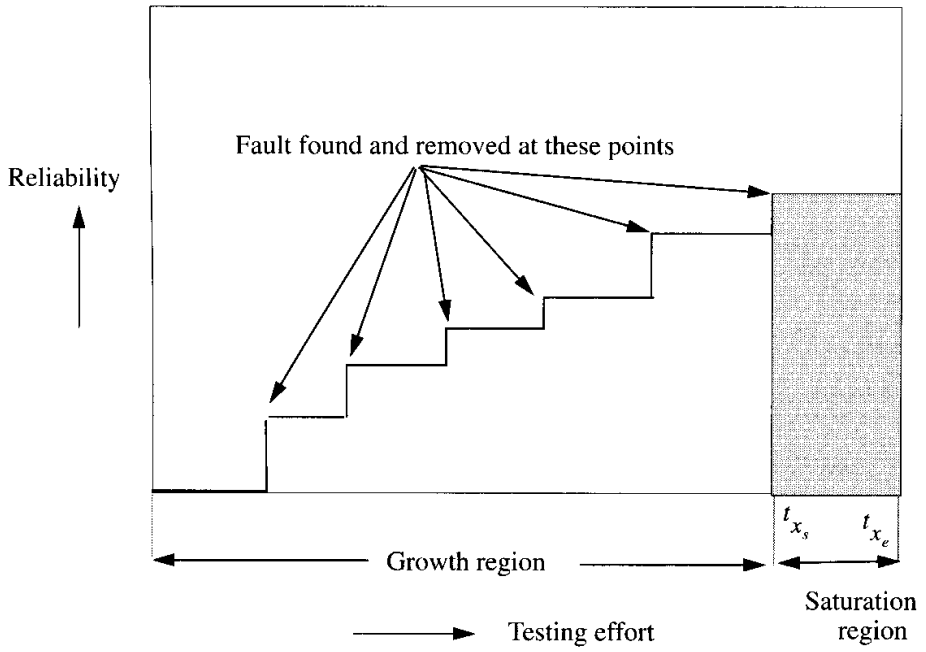
reliability. In Fig. 13.2, $\overline{R_b}$ is the estimate of the true reliability R_b at the end of functional testing.

The above reasoning applies to other phases of testing as well when white-box testing methods are being used. In each case, there exists a period of testing when $\overline{R}$ increases even though R remains fixed. Such an increase can lead to significant overestimates, as shown in Fig. 13.2.

Figure 13.2 indicates that R is a monotonically increasing function of testing effort in the growth region. This may not be always true. In general, the growth region will appear as in Fig. 13.3. Thus, for example, if t denotes CPU time spent in executing P, then R will grow during periods when faults are found; otherwise it will remain constant. This stepwise rise of R will cause $\overline{R}$ to fluctuate and increase stochastically, as dictated by the underlying model.

13.4.8 Incorporating coverage in reliability estimation

As mentioned earlier, during black-box testing we execute the system against test cases developed using the operational profile. Each test case has an effect on the *coverage* of various program elements. As a simple example, when program P is executed against test case $t_i, i \geq 1$, it may cause some statements to be executed for the first time, thus increasing the statement coverage. Below we outline an approach that



t_{x_s} : Start of saturation region for testing technique x

t_{x_e} : End of saturation region for testing technique x

Figure 13.3 A realistic growth region.

makes use of coverage measures in estimating reliability. Our approach does not necessarily entail development of test cases using coverage measures; however, it calls for measurement of coverage during testing.

The time-structure-based approach outlined below is a simple extension of the the existing approach based on time-domain models. We emphasize that the use of time-structure-based approach does not require a change in the method used for the development of a test set; it does require the measurement of coverage such as block, decision, or data flow coverage. The pure structure-based models remove the notion of time in estimating the reliability and lead to another approach to reliability estimation. It is not yet clear which approach is the best to use in practice.

13.4.9 Filtering failure data using coverage information

We now describe an approach to incorporate coverage information in estimating software reliability. We begin by defining the notion of *useful* testing effort. A testing effort E_k is useful if and only if it increases some type of coverage. Note that the definition of usefulness does not specify which coverage should be increased for a test effort to be useful.

We might argue that in practice every test case, against which P has not been executed before, is useful. This argument is acceptable in accordance with our definition of usefulness if input domain coverage is considered to be one type of coverage. We know that there exist disjoint subsets D_i, $1 \leq i \leq n, n \geq 0$, of D, known as *partitions,* such that $\cup D_i = D$, that will cause P to behave identically. Thus, testing P on one element of D_i is equivalent to testing P on all elements of this partition. The problem, however, is that we cannot compute these partitions a priori. Instead, we rely on various testing methods to provide us with the relevant partitions. Therefore, we assume that input space coverage is *not* one of the coverage types to be considered in determining whether an effort is useful or not. Later we will explain how this assumption affects reliability estimates based on time/structure models.

We have already defined three types of coverages, namely, decision, data flow, and mutation. To illustrate the notion of usefulness, suppose that the first $k - 1$ test cases have resulted in a decision coverage of 35 percent. Now if the kth test case increases this coverage to, say, 40 percent, then we say that it is useful. In case the CPU time spent is the measure of testing effort, then an execution of P that causes an increase in decision coverage results in useful testing effort. Note that there are several ways of measuring structural coverage. It is not clear which is the best and should be used here. We return to this question later, in Sec. 13.4.10.

The effort E_k defined in Eq. (13.2) consists of one or more atomic efforts e_i. However, an e_i may be useless. To account for such useless efforts, which may bias the interfailure effort, we define

$$E_k^c = \sum_{i=l_1}^{l_2} \sigma e_i \tag{13.6}$$

where l_1 and l_2 are as in Eq. (13.2) and σ is the *compression ratio.* The quantity σ can be defined in several ways. Below we provide a simple definition. We consider alternate definitions in Sec. 13.4.10.

$$\sigma = \begin{cases} 1 & \text{if } e_i \text{ increases coverage} \\ 0 & \text{otherwise} \end{cases} \tag{13.7}$$

The use of compression ratio compresses the interfailure effort, E_k to E_k^c by ignoring the atomic effort that has been found useless. This process is illustrated in Fig. 13.4. Along the thick horizontal line, the testing effort is indicated. The leftmost upward-pointing arrow indicates the instant when testing began; subsequent upward arrows mark failure points. Two consecutive downward arrows bracket the atomic effort. The first atomic effort is bracketed by the leftmost upward arrow and the first downward arrow. The sequence of total effort spent between successive failures is indicated by the sequence of shaded boxes labeled "Observed." The sequence of shaded boxes, labeled "Useful" just below this line indicates the *filtered* effort data obtained by applying the compression ratio to the observed data.

The filtered effort data can now be used in any of the existing time-based models. Thus, for example, if the Musa model is being used to predict reliability, then the filtered data can serve as the modified sequence of interfailure times. If $\overline{R}_m$ and $\overline{R}_f$ are reliability estimates generated by the Musa basic model using, respectively, the original and filtered interfailure times, then $\overline{R}_f < \overline{R}_m$. The example below illustrates this relationship. Thus, filtering leads to a more realistic reliability estimate than the approach that uses unfiltered data. The filtered data can also be applied to any of the other models, such as the Goel-Okumoto model [Goel85].

Example 13.1 To show that filtered interfailure data result in more realistic reliability estimates, we generated hypothetical data corresponding to E_k. These data were filtered and reliability estimates computed using the Musa basic execution time model. The method for doing so is described below. Here we assume that the effort is measured in terms of CPU time.

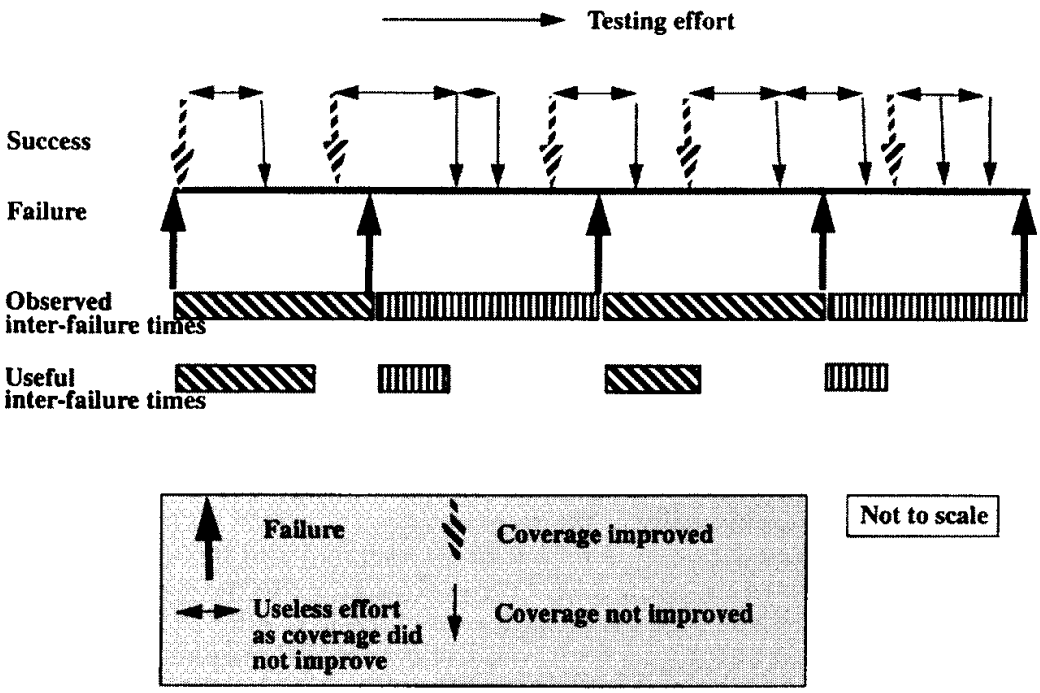


Figure 13.4 Filtering failure data using coverage information.

1. Assume that the per-fault hazard rate $\Phi = 0.05$ and the total number of expected failures $V = 100$.
2. The number of failures experienced by time t, denoted by $u(t)$ is computed as

$$u(t) = V(1 - e^{(-\Phi t)}) \tag{13.8}$$

3. The time to the ith failure, t_i, is computed from Eq. (13.8) as

$$t_i = \ln(V/(V - u))/\Phi \tag{13.9}$$

4. The failure interval between $(i - 1)$th and ith failure, Y_i, is $t_i - t_{i-1}$.
5. However, Y_i, as computed above, is monotonically increasing due to the use of Eq. (13.9). To obtain more realistic interfailure time data, we use Y_i as the mean of an NHPP process and generate the interfailure times. Let r be a uniform random number, $0 < r < 1$. We compute the interfailure times from Y_i as

$$E_i = Y_i \ln(1/(1 - r)) \tag{13.10}$$

6. Assume that P requires a constant time to execute on each test case. Let this time t_P, arbitrarily chosen, be 0.01 time unit. We then obtain the number of test cases n_i used in the ith failure interval as E_i/t_P.
7. Let r' be a uniform random variate, $0 < r' < 1$. The probability that a test case used at time t did not improve coverage is $1 - e^{(-0.01tr')}$. Using this information we identify which test cases are useless and hence contribute to useless effort. This information is used for compressing E_i to E_i^c.
8. Compute the new failure times $t_k = \Sigma_{i=1}^{k} E_k^c$, $1 \le k \le 100$.
9. Compute the new per-fault hazard rate Φ' for the uncompressed data and Φ'' from the compressed data. Maximum likelihood estimation is used in both cases.

10. Apply the Musa basic model to the uncompressed and compressed interfailure times to obtain the two reliability estimates, $\overline{R}(x/t)$ and $\overline{R}^c(x/t)$, respectively.

Using the uncompressed and compressed interfailure time data, we computed various reliability estimates. Figure 13.5 shows $\overline{R}(0.01 \mid t)$ and $\overline{R}^c(0.01 \mid t)$, i.e., the reliability for an exposure period of 0.01 time units at time t. Notice that both the estimates are almost equal but in all cases $\overline{R}^c(0.01 \mid t) \leq \overline{R}(0.01 \mid t)$. Over the entire time duration for 100 failures, we get $0.784 \leq \overline{R}(0.01 \mid t) \leq 0.999$ and $0.783 \leq \overline{R}^c(0.01 \mid t) \leq 0.998$.

The difference in the two estimates is significant in Fig. 13.6 where the exposure period is 10 time units. For this data set we obtained $0.0 \leq \overline{R}(10 \mid t) \leq 0.65$ and $0.0 \leq \overline{R}^c(10 \mid t) \leq 0.445$.

Reliability estimates were also obtained by fixing t and varying the exposure time. Figure 13.7 shows the estimates $\overline{R}(x \mid t)$ and $\overline{R}^c(x \mid t)$ where the values of the current time were set arbitrarily to the time at which the 84th failure occurred. The range of estimates obtained is $0.01 \leq \overline{R}(x \mid t) \leq 0.995$ and $0.003 \leq \overline{R}^c(x \mid t) \leq 0.993$. Figure 13.8 shows the estimates obtained by fixing t to the time when the 99th failure occured. The range of estimates obtained is $0.562 \leq \overline{R}(x \mid t) \leq 0.999$ and $0.343 \leq \overline{R}^c(x \mid t) \leq 0.998$. These data, and the data mentioned above, indicate

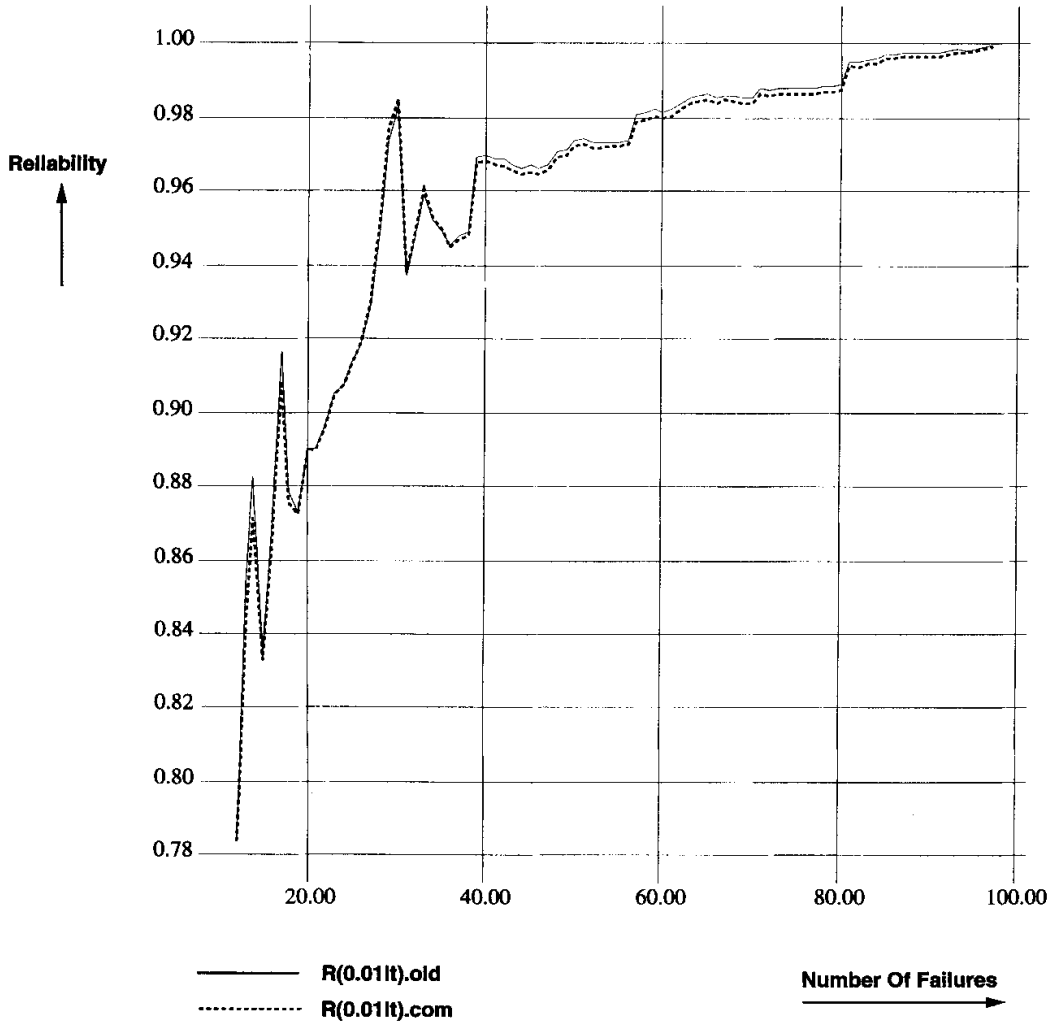


Figure 13.5 Variation in the reliability estimates with number of failures, $\overline{R}(0.01 \mid t)$.

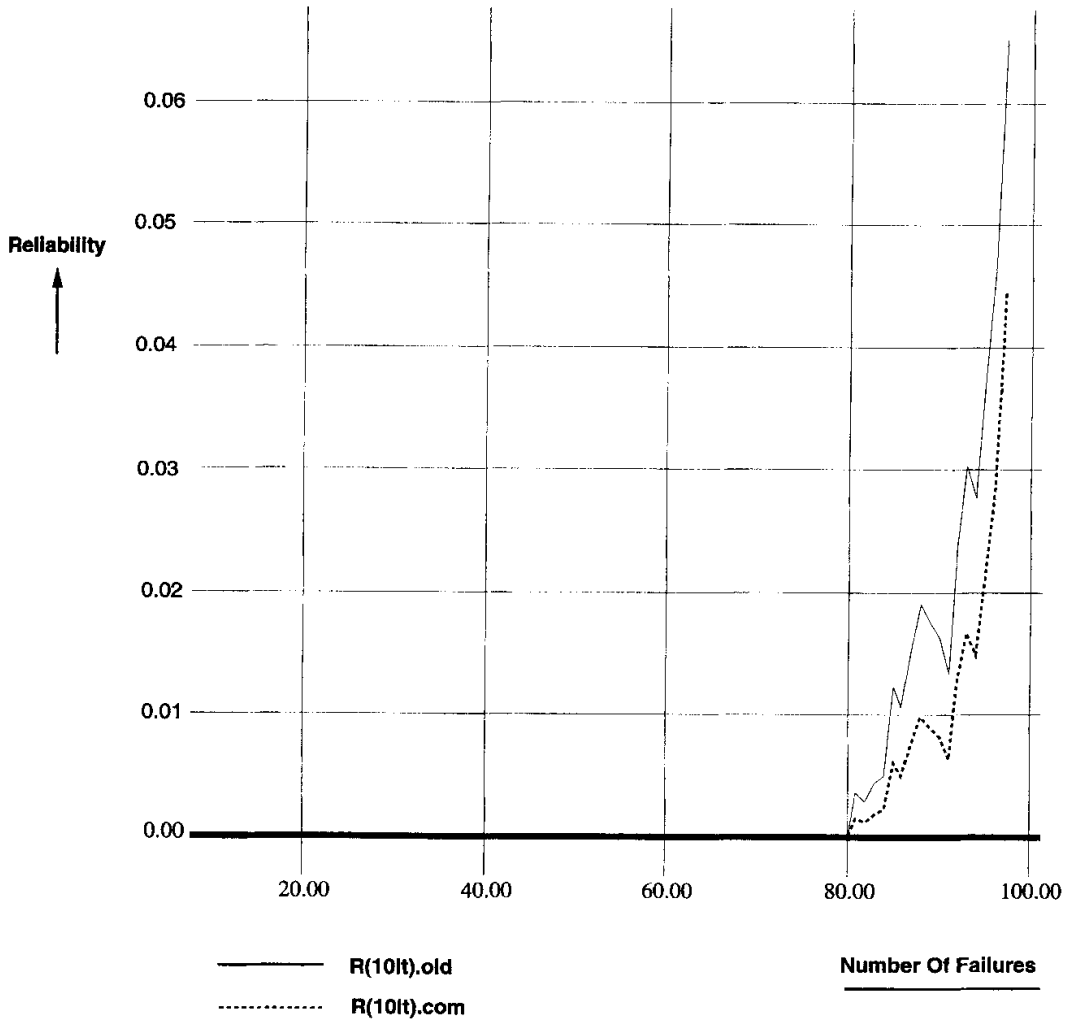


Figure 13.6 Variation in the reliability estimates with number of failures, $R(10 \mid t)$.

that the estimates using the time structure model are conservative, as indicated by the ratio $\overline{R}(x \mid t)/\overline{R}^c(x \mid t)$, which ranged from 1 to 3.33.

13.4.10 Selecting the compression ratio

The notion of useless effort stems directly from the fact that the input domain of P can be partitioned into disjoint subdomains. Once such a partition is available, it is necessary to select just one test case from each subdomain. If two test cases are selected from one partition then one of them leads to useless effort. Even though domain partitioning is possible in theory, in practice it is difficult to determine such partitions for nontrivial programs. The white-box testing methods provide, at best, an approximation to such a partition. Thus, for example, if a test set covers a decision, we assume that any other test case that once again exercises the same decision is useless. It is easy to show by examples, and has also been shown empirically [Howd80], that such a test case may indeed reveal faults. This implies that what we consider as a

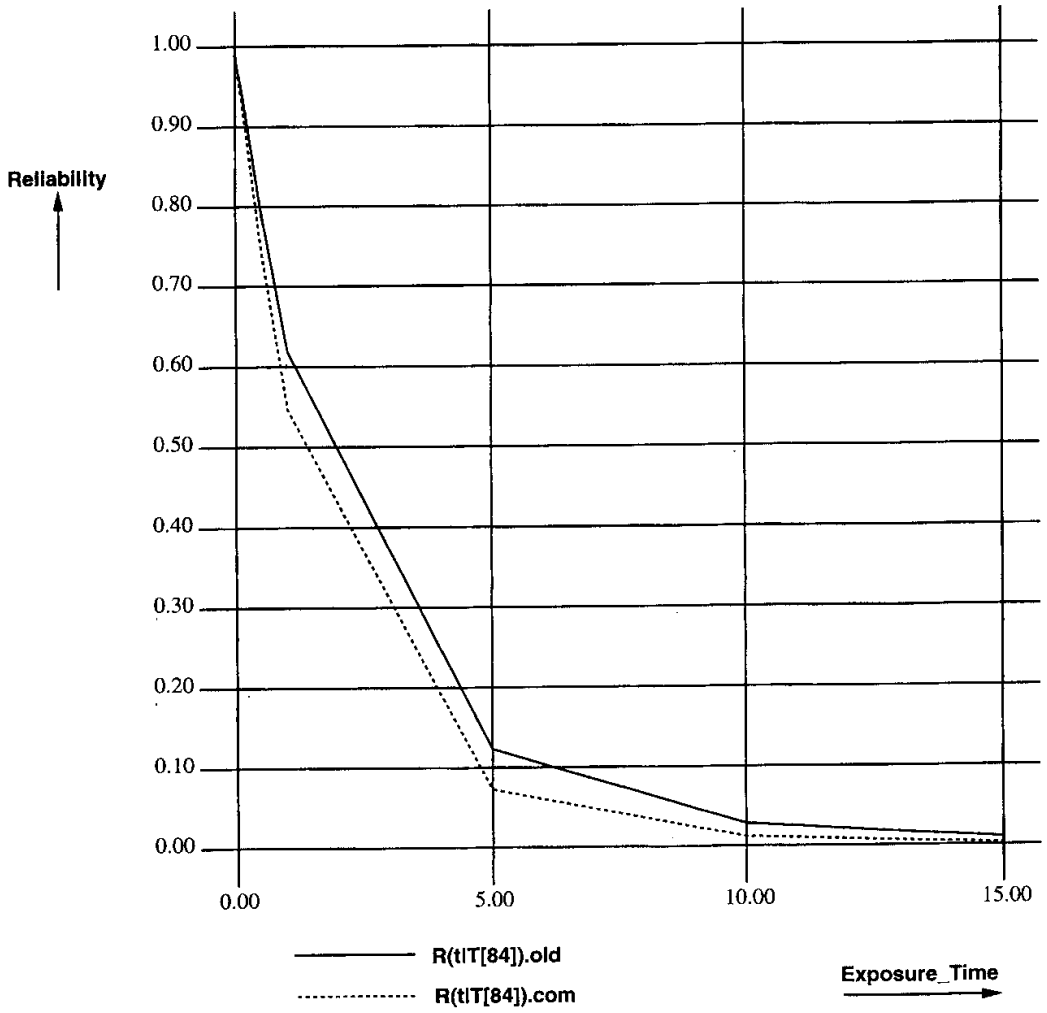


Figure 13.7 Variation in the reliability estimates with the exposure time, $R(x \mid 84)$.

test case amounting to useless effort may indeed be a useful test case that, when run successfully on P, has shown the nonexistence of a fault. Such a test case should therefore improve our reliability estimate. However, other than structural or mutation coverage, we do not have any other criterion to test the utility of a test case.

The above reasoning leads us to reconsider the definition of σ as given in Eq. (13.7). Toward this end, we make the following observations:

1. It is more likely for a test case to increase a coverage measure during the initial phases of testing than later. Thus, as coverage increases, it becomes increasingly difficult to construct a test case that will further increase coverage.
2. Once one or more coverage criteria have been satisfied, the likelihood of a test case, selected randomly, increasing coverage according to any remaining criteria, decreases.

As an example, suppose that T_d is a test set that covers all decisions but does not provide 100 percent data flow coverage. If a test case $d \notin$

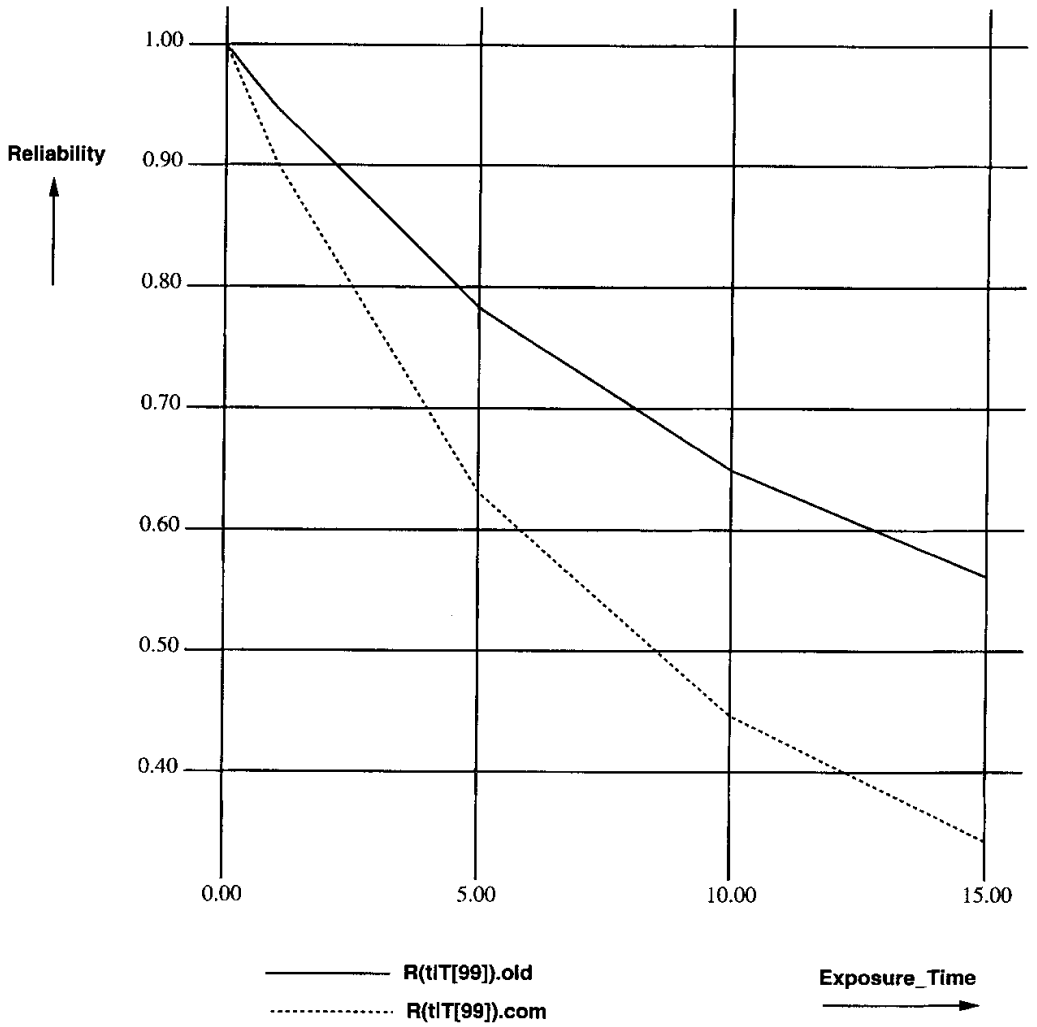


Figure 13.8 Variation in the reliability estimates with the exposure time, $R(x \mid 99)$.

T_d is now selected randomly from the input domain, the likelihood of d increasing data flow coverage is less than its likelihood of increasing decision coverage when 100 percent decision coverage was not reached.

The above observations lead us to hypothesize that a test case d_1 may be more *important* than another test case d_2. We can modify Eq. (13.7) to account for this importance. The compression ratio can be made effort dependent. Thus, $\sigma \equiv \sigma(t)$ and $\sigma(t_1) < \sigma(t_2)$ if $t_1 < t_2$. Such a definition of σ will ensure that test cases that increase coverage during the later phases of testing get more importance than those that increase coverage during the early phases. Experiments carried out using another filtering method are described in [Chen94a].

13.4.11 Handling rare events

For a given test case $d \in D$, the failure of P on d, indicated by incorrect $P(d)$, is considered a rare event if Prob $\equiv P\{P(d)$ is incorrect$\} < \varepsilon$ for some arbitrarily small ε. If the operational profile is used during test-

ing, then Prob is dependent on the profile itself. If the profile does not contain d, and hence P was most likely not exercised on d, then the failure might occur after a significant amount of effort has been spent during program operation. Such a failure is likely to be considered as a rare event arising due to the occurrence of a possibly rare input d during operation. If the profile is not used during testing, then Prob is the same as the probability of occurrence of d during operation.

In either of the two cases mentioned above, it seems impossible to determine such a test case during testing. This is specially true when only a negligible fraction of the input domain is accounted for during testing. Inclusion of coverage in reliability estimation helps decrease the likelihood of a failure-generating input arising in operation. Such a decrease takes place in two ways.

First, as mentioned above, coverage-based reliability estimates have been found to be more realistic compared to the ones that ignore coverage data. This is expected to lead to increased testing effort to raise the estimated reliability up to a cutoff level that may have been set by the management. Second, an examination of coverage helps the tester construct new test cases in addition to the ones constructed during functional testing. Such test cases are likely to reveal faults in code that remained uncovered during functional testing, based perhaps on the operational profile. Thus, failures that may have proved to be rare events during operation may in fact occur during testing.

13.5 A Microscopic Model of Software Risk

In this section we present a *risk model* for computing and/or interpreting the reliability numbers in conjunction with coverage data obtained during testing.

13.5.1 A testing-based model of risk decay

The computational basis for the risk model is the control and data flow measures. For purposes of discussion below we consider measures supported by ATAC, a data flow testing tool which reports various data flow coverages for a given test set. To fully capture the intuitive notion of the *risk of untested code,* other testing-based notions of risk that are intuitively independent of the data flow concept might also be considered. One such notion is provided by *mutation* testing [Choi89] and another by *domain* testing. By counting risk on three intuitively independent testing scales we might capture much of the notion of risk. For instance, during testing with ATAC we may be able to reduce the data flow risk of tested code to zero. The remaining mutation and domain risk would now provide a measure of residual testing risk, which could be taken into account in reliability and safety assessment.

Data flow testing measures the adequacy of a set of tests according to how well the tests exercise the statements, branches, and variable definition/use pairs in a program. For a given program P in the C language and a test set T for use with P, ATAC reports, among other data, a coverage score like

```
% blocks       % branches    % All-Uses
33(169/516)    21(86/414)    15(277/1852)
```

The above scores can be interpreted to mean that T exercised 33 percent of the 516 basic blocks, 21 percent of the branches, and 15 percent of the variable definition/use pairs in the program P. Having executed P on each element of T, one might view the above score as a multifaceted risk profile of P. A simple interpretation of these measures, assuming that all discovered faults have been repaired, is that by running the tests in T, we have reduced the risk associated with the untested blocks of P by 33 percent, that associated with untested branches by 21 percent, and of untested variable definition/use pairs by 15 percent. If another test set T_1 yields the coverage score

```
% blocks        % branches      % All-Uses
100(516/516)    100(414/414)    100(1852/1852)
```

we might say that P is risk-free when measured by block, branch, and all-uses testing. The conclusion of this sort of simple "risk" interpretation of coverage testing is that a set of tests that visit all blocks, branches, and def/uses in P eliminates all risk associated with P.

We now examine the notion of *risk* that originates from data flow coverage testing in the following examples. For simplicity, we restrict our examples to consideration of coverage of basic blocks; the model can be used to compute risk by including other data flow coverage statistics as well.

13.5.2 Risk assessment: an example

The program of Fig. 13.9 contains 12 basic blocks. These basic blocks are the indivisible units of control execution; every statment or expression in a basic block will be executed if any is executed. The simplest testing-based notion of risk is: *basic blocks (or statements) which remain untested are risky.* This dictum is simple, easily assessed, but seldom observed. Very few software organizations require every statement of code under test to actually be executed by some test. Occasionally, such an omission is justified. Some code is defensive code that cannot be executed, and some code can be executed only under conditions impossible to reproduce in a testing environment, but, often, statement coverage is simply not assessed.

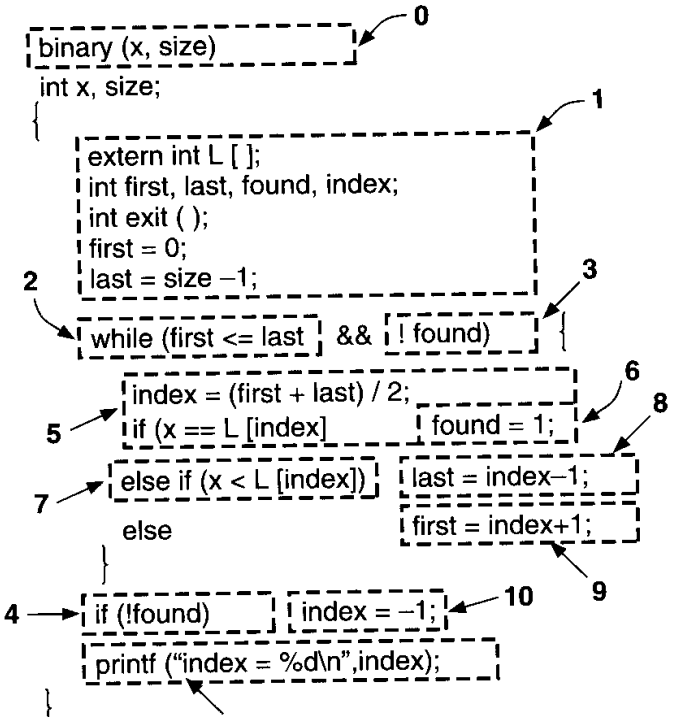


Figure 13.9 Basic blocks of a simple program.

Typically, we would run a set of tests like those displayed in Table 13.2.

Subsequent to running these tests we can calculate the coverage of the tests. Test t.1 yelds the following coverage:

```
> atac -s -n t.1 binary.atac
% blocks    % decisions % C-Uses    % P-Uses
----------- ----------- ----------- -----------
83(10/12)   70(7/10)    70(7/10)    61(17/28)  == total ==
```

Tests t1, t2, t3, and t4 together yield the following:

```
> atac -s -n t.[1-4] binary.atac
% blocks    % decisions % C-Uses    % P-Uses
----------- ----------- ----------- ------------
92(11/12)   90(9/10)    80(8/10)    71(20/28)  == total ==
```

The set of all 16 tests give the following coverage:

```
> atac -s -n t.* binary.atac
% blocks    % decisions % C-Uses    % P-Uses
----------- ----------- ----------- ------------
100(12)     100(10)     100(10)     93(26/28)  == total ==
```

TABLE 13.2 Sixteen Tests for binary.c

Variables	x	L[0]	L[1]	L[2]
Test t.1 values	0	0	1	2
Test t.2 values	0	0	1	3
Test t.3 values	0	0	2	3
Test t.4 values	0	1	2	3
Test t.5 values	1	0	1	2
Test t.6 values	1	0	1	3
Test t.7 values	1	0	2	3
Test t.8 values	1	1	2	3
Test t.9 values	2	0	1	2
Test t.10 values	2	0	1	3
Test t.11 values	2	0	2	3
Test t.12 values	2	1	2	3
Test t.13 values	3	0	1	2
Test t.14 values	3	0	1	3
Test t.15 values	3	0	2	3
Test t.16 values	3	1	2	3

Table 13.3 gives the number of visits to each block by each of the 16 tests as computed by ATAC.

Based on research in the fault-detection ability of coverage measures, we assume that statement coverage alone is not a good measure of risk reduction [Girg86]. Data flow testing measures the coverage of *branches* and various *definition/use* relationships [Horg91]. Covering these more complex aspects of code can be shown to reduce the exposure of code to faults, and thus risk. For instance, Fig. 13.10 displays

TABLE 13.3 Test Visit to Blocks

Blocks	0	1	2	3	4	5	6	7	8	9	10	11
t.1 visits	1	1	3	3	1	2	1	1	1	0	0	1
t.2 visits	1	1	3	3	1	2	1	1	1	0	0	1
t.3 visits	1	1	3	3	1	2	1	1	1	0	0	1
t.4 visits	1	1	3	2	1	2	0	2	2	0	1	1
t.5 visits	1	1	2	2	1	1	1	0	0	0	0	1
t.6 visits	1	1	2	2	1	1	1	0	0	0	0	1
t.7 visits	1	1	3	2	1	2	0	2	1	1	1	1
t.8 visits	1	1	3	3	1	2	1	1	1	0	0	1
t.9 visits	1	1	3	3	1	2	1	1	0	1	0	1
t.10 visits	1	1	3	2	1	2	0	2	1	1	1	1
t.11 visits	1	1	2	2	1	1	1	0	0	0	0	1
t.12 visits	1	1	2	2	1	1	1	0	0	0	0	1
t.13 visits	1	1	3	2	1	2	0	2	0	2	1	1
t.14 visits	1	1	3	3	1	2	1	1	0	1	0	1
t.15 visits	1	1	3	3	1	2	1	1	0	1	0	1
t.16 visits	1	1	3	3	1	2	1	1	0	1	0	1

data from an experiment in which our colleagues compared the statement coverage of unit tests for 28 modules of a single system to the number of system test faults found for each module. There is a clear relationship between high statement coverage in unit testing and low system test faults [Dala93].

13.5.3 A simple risk computation

Our model computes the risk of a code fragment by computing the risk of the constituent *testable attributes* of the code fragment. For the program `binary.c` these attributes are the 12 basic blocks, the 10 decisions, the 10 c-uses, and the 26 p-uses. We consider each of these attributes as carrying a unit of *static risk*. Each of these attributes is considered as a *receptor* of a potential defect. Of course, we do not know whether a given c-use, for example, is defective or not, but we do associate risk with it to account for the possibility of it being defective. Another way of looking at static risk in our model is to view it as a method of disproportionally distributing expected defects throughout the code. Using $\alpha_k(l_i), 1 \leq k \leq N$ to represent the static risk for all attributes of type k (e.g., c-uses) constituent in *locus* l_i (e.g., a line of code), we compute the static risk $\rho(l_i)$ for a given program for locus l_i as

$$\rho(l_i) = \sum_{k=1}^{N} \alpha_k(l_i) \tag{13.11}$$

Considering ζ_k defects per attribute as *prior information*, $\zeta\rho(l_i)$ is defined to be the *expected defects on prior information* for locus l_i:

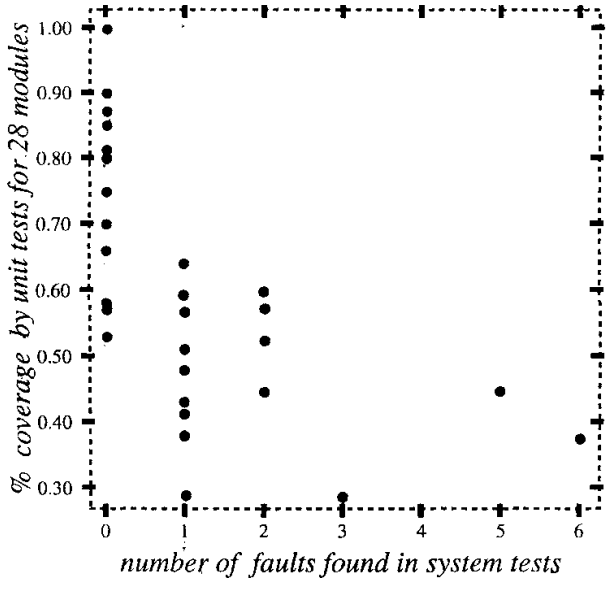


Figure 13.10 Relationship of unit coverage testing to system test faults for one system.

$$\zeta\rho(l_i) = \sum_{k=1}^{N} \alpha_k \zeta_k(l_i) \tag{13.12}$$

Using $\alpha_1(l_i)$ to represent blocks, $\alpha_2(l_i)$ decisions, $\alpha_3(l_i)$ c-uses, and $\alpha_4(l_i)$ p-uses, the static risk, $\rho(l_i)$, for locus l_i of `binary.c` is given by

$$\rho(l_i) = \alpha_1(l_i) + \alpha_2(l_i) + \alpha_3(l_i) + \alpha_4(l_i)$$

The expected defects on prior information is given by

$$\zeta\rho(l_i) = \alpha_1\zeta_1(l_i) + \alpha_2\zeta_2(l_i) + \alpha_3\zeta_3(l_i) + \alpha_4\zeta_4(l_i)$$

To be more concrete, suppose from prior project experience we believe that `binary.c` has 0.5 faults per 1000 noncommented source lines. Then, as `binary.c` has 17 noncommented source lines, it should have 8.5×10^{-3} faults. If we weigh all testable attributes equally (58 in this case), then there are approximately 1.47×10^{-4} faults per attribute. Relating that to a line of code is done through basic blocks. Line 13 of `binary.c`, `"else first = index + 1"`, is involved in 1 basic block, 0 decisions, 1 c-use, and 5 p-uses. Therefore, we have

$$\alpha_1(\text{line 13}) = 1 \qquad \alpha_2(\text{line 13}) = 0,$$

$$\alpha_3(\text{line 13}) = \tfrac{1}{2} \qquad \alpha_4(\text{line 13}) = \tfrac{5}{3}$$

As each c-use participates in 2 basic blocks and each p-use participates in 3 basic blocks, we assign ½ of a c-use and ⅓ of a p-use for each occurrence in line 13. Then the static risk, ρ(line 13), for locus line 13 of `binary.c` is given by

$$\rho(\text{line 13}) = 1 + 0 + \tfrac{1}{2} + \tfrac{5}{3} = 3.17$$

As ζ_k is 1.47×10^{-4} per attribute in our example, the expected defects on prior information is given by

$$\zeta\rho(\text{line 13}) = 1.47 \times 10^{-4} \times 3.17 = 4.66 \times 10^{-4}$$

The expected defects on prior information for line 13 reflects our view that testable attributes are markers for defects. Therefore, when we calculate the static risk for line 13 as a portion of the static risk for the entire program, we distribute the expected defects of `binary.c` according to that risk. Thus, 5.5 percent (3.17/58) of the total static risk and the expected number of defects of `binary.c` are attributed by our model to line 13 of `binary.c`.

13.5.4 A risk browser

We have developed a tool named Risk Browser that allows a user to browse through the source code of a program displaying the risk associated with individual parts of the code. As an example, Fig. 13.11 shows the detailed display of static risk for the program `binary.c`. That is, the risk as measured by our model based upon the distribution of testable data flow attributes to the individual lines of `binary.c` before *any* tests are run. This fine-grained report of risk forms a baseline of the relative risk of the individual lines. Surprisingly, the 10th and 11th nonblank lines are considered most risky because they involve the highest number of data flow relations when compared with other lines in the code. In this respect, our static measure of risk is considerably different than control-flow-based measures such as cyclomatic complexity.

In a large software system the risk browser can present cumulative risk for various units of the system. In Fig. 13.11, the background window shows the relative cumulative static risk for the two files main.c (not presented here) and `binary.c`. As main.c is a simple driver pro-

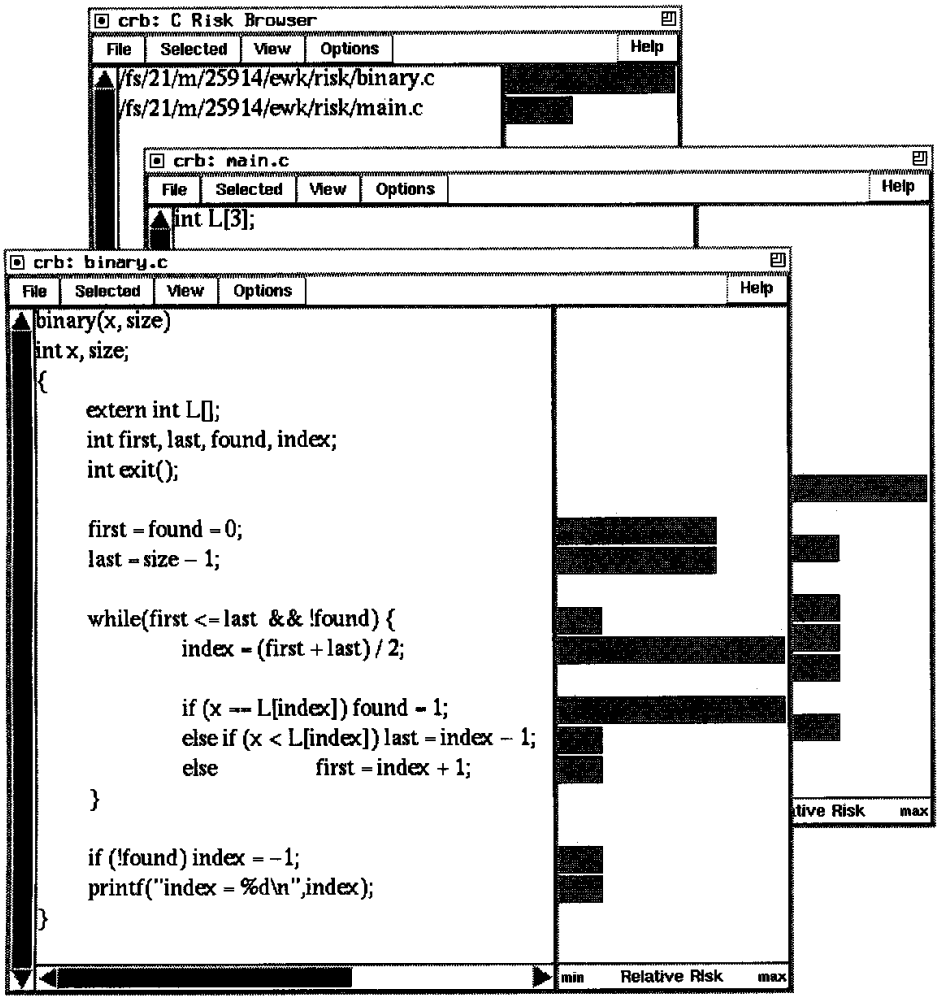


Figure 13.11 Static risk for `binary.c`.

gram, `binary.c` is shown to be considerably more risky. In all displays of risk we normalize against the most risky component of the display window. So lines 10 and 11 in the foreground window are shown with maximal risk, while `binary.c` in the background window is displayed with maximum risk.

Static risk is the start, and risk decays as testing progresses. Figure 13.12 shows the interaction of the risk browser and ATAC as the programmer attempts to reduce dynamic risk. The left foreground window shows the dynamic risk for `binary.c` after t.1 through t.8 have been run. The top right foreground window shows an ATAC display of uncovered blocks and the bottom foreground window shows an ATAC display of uncoverd decisions which remain in `binary.c` at this point in testing. The programmer crafts tests t.9 through t.16, which test these uncovered attributes, and the result is the final display of risk in Fig. 13.13. Here we see the original static risk (outlined) and the residual risk (darkened).

13.5.5 The risk model and software reliability

As ATAC collects visitation rates by test for each testable attribute, a more complex model of risk is possible. Rather than allowing the risk associated with an attribute to vanish once covered, we can allow the risk to asymptotically decay as visitation increases. We can construct

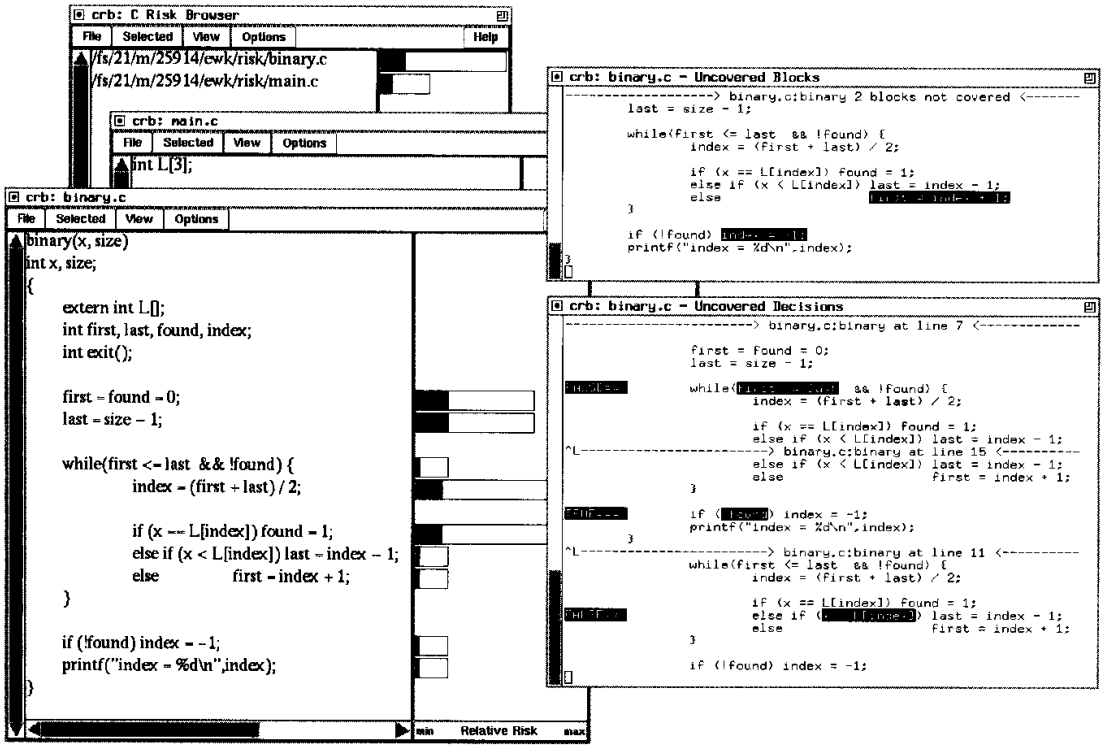


Figure 13.12 Dynamic risk for `binary.c` after execution on t.1 through t.8.

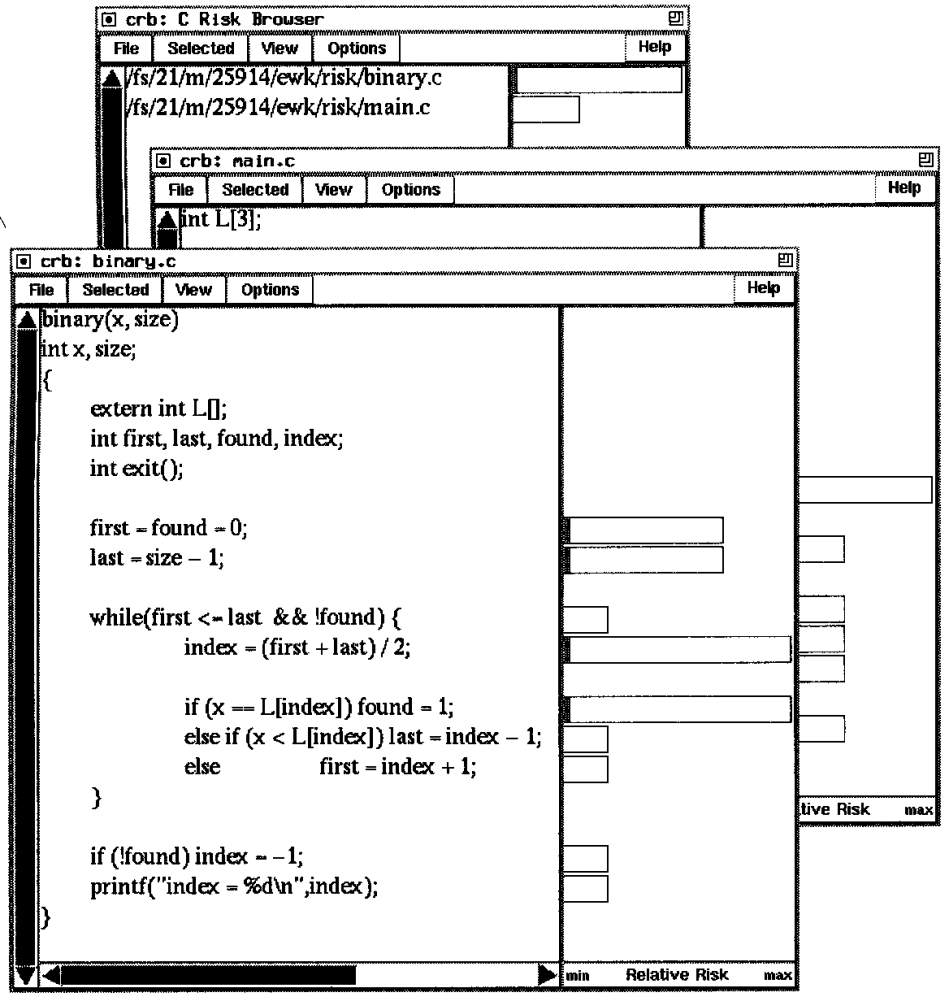


Figure 13.13 Dynamic risk for `binary.c` after t.1 through t.16.

confidence growth models similar to reliability growth models. Currently, the pragmatic value of such models is not known.

A more important relationship between coverage testing and reliability modeling involves the saturation effect explained earlier. Suppose that we calculate the reliability of `binary.c` based on the sequence of tests t.1 through t.16. This would be a trivial exercise, but we can ask "Should all the tests be counted as a measure of effort?" Surely, a sequence of 16 consecutive runs of t.1 would not count as 16 units of time or effort. An exact analysis using ATAC tells us that tests t.1, t.2, t.3, and t.8 define exactly the same computation paths in `binary.c`. The same is true of the sets {t.4}, {t.5, t.6, t.11, t.12}, {t.7, t.10}, {t.13}, and {t.9, t.14, t.15, t.16}. Thus, if our notion is that only testing which reveals something new about a program should be used in measuring reliability growth, we could restrict the reliability calculation to {t.1, t.4, t.5, t.7, t.9}. Moreover, if our notion is that only testing which reveals new coverage should be considered in reliability assessment, then only {t.4, t.13, t.16} need be considered. ATAC permits such measurements.

There is a problem of scale in these sorts of calculations. ATAC is used to monitor moderately large programs (up to 100,000 lines) during product testing. However, run-time overhead is sometimes several times the ordinary runtime. The data collection for exact analysis of path redundancy among tests is an additional burden. For large systems, the measurements required for our models may require off-line reprocessing of the tests so as not to interfere with the normal testing process.

13.6 Summary

In this chapter we have pointed to the problems encountered in reliability estimation in the presence of an inaccurate operational profile. To overcome these problems we have described two different methods of accounting for code coverage in software reliability estimation. The first method makes use of code coverage data to filter the failure data input to time-domain models. Any of the traditional time-domain models is then used to obtain reliability estimates. The coverage information can be obtained during the system test phase. Experiments conducted on small programs and on large simulated programs show that this method leads to more realistic estimates when compared with estimates obtained using the unfiltered data.

In the second method, code coverage measures and the program structure are used to assess the risk associated with a program. Highly reliable software is considered as software associated with low risk. Given visitations and coverage data for a program and a set of tests, a method of calculating risk has been described.

Problems

13.1 *a.* How does white-box testing differ from black-box testing?

b. What in your experience is the likelihood of obtaining 100 percent statement coverage from a test set derived using black-box testing?

c. Is it possible to test all functions from a test set that is derived to obtain 100 percent statement coverage?

d. For some program P, a test set T is adequate with respect to the statement coverage criterion. Give an example P for which T is not adequate with respect to the decision coverage criterion.

13.2 Consider the following program that inputs two integers a and b and computes y.

```
begin
    integer x,y,a,b;
    input(a,b);
    x=0; y=10;
```

```
    if a>0 then
                n=1
           else
                n=2
    endif;
    if b>0 then
                y=(n+1)*a
           else
                y=(n-1)*b
    endif;
    output (y);
end
```

a. Construct a test set T_s adequate with respect to the statement coverage criterion.

b. Construct T_d by adding test cases to T_s such that T_d is decision adequate.

c. Construct T_{pc} by adding test cases to T_s such that T_{pc} is p-use and c-use adequate.

d. A test set T is considered minimal with respect to some criterion C if removal of any test case from T will cause the remaining test set to be inadequate with respect to C. Which of the test sets constructed above are minimal?

13.3 *a.* Assume that program P' is constructed from P of Prob. 13.2 by replacing the statement $n = 2$ by $n = 3$. Consider P' to be an erroneous version of P. Which of the test sets constructed in Prob. 13.2 will cause P' to fail?

b. Let T be some test set that is adequate with respect to the statement coverage criterion for P'. Is T guaranteed to reveal the error in P' by causing P' to fail?

c. How does your answer to item *b* change if the statement coverage criterion is replaced by decision coverage? p-use coverage? c-use coverage?

d. Suppose that the input domain of P consists of all values of a and b such that $-5 \le a < 5$ and $-5 \le b < 5$. Let (a_1,b_1) be a randomly selected pair from the input domain. Let P' be executed on (a_1,b_1). What is the probability that P' will fail? How does this probability change as the range of a and b (i.e., the size of the input domain) increases?

13.4 For some user of program P' of Prob. 13.3, the operational profile has been specified as shown in Fig. 13.1. Suppose that P' is tested by randomly selecting a pair from the input domain using the operational profile given in Table 13.4.

a. With what probability will P' fail?

b. How does the probability in item *a* change as the probability of a test case, which satisfies $a < 0$ and $b < 0$, reduces?

c. Suppose that each time P' fails, the user incurs a cost C. What approximate cost will the user incur after 100 executions of P'?

TABLE 13.4 Sample Operational Profile for Prob. 13.4

	Input	Probability
1	$(a = 0, b > 0)$	0.1
2	$(a = 0, b = 0)$	0.005
3	$(a > 0, b > 0)$	0.7
4	$(a > 0, b = 0)$	0.1
5	$(a < 0, b < 0)$	0.05
6	All others	0.045

d. Suppose that P' was tested with respect to the operational profile given in Table 13.4 and that the error in P' was not revealed. P' is now delivered to a user whose operational profile is different from the one given in Table 13.4. The difference is in entries 3 and 6. These entries for our user are

$(a > 0, b > 0)$	0.4
All others	0.345

Recompute your answer to item *c* for the modified operational profile.

e. Now suppose that P' was tested to achieve 100 percent statement coverage. How will your answer to item *d* change? For decision coverage? For *p*-use and *c*-use coverage?

13.5 *a.* Why and how does the saturation effect affect the reliability estimate of a program?

b. Why is it that the use of code coverage in reliability estimation using random testing is likely to improve the estimates?

c. When using a "white-box" based approach to reliability estimation when is a test case considered useful?

d. Suppose that while testing for reliability measurement, we measure statement coverage. Now, suppose that a test case t does not increase the statement coverage. Will t be considered useful? Will it be considered useful if some other coverage were measured?

13.6 During the execution of a program P, we say that a rare event has occurred if P has been executed on an input that occurs with a very low probability. Which of the two testing methods—random or coverage-based testing—is more likely to cause this event to occur during testing? Provide a sample program to illustrate your answer.

13.7 Let P be a program with three features: f_1, f_2, and f_3. The program consists of 300 lines of executable source code. We test P by providing exactly three

test cases, one to exercise each feature. Do you expect these test cases to provide 100 percent statement coverage? How does your answer change as the size of the program is increased and the number of features remains fixed? Explain your answer.

13.8 *a.* What is the basic difference between the approaches outlined in the chapter for reliability estimation using code coverage versus the risk decay model?
b. Why is it that statement coverage alone is not a good measure of risk reduction?
b. How is the static risk measure affected by increase in code coverage?

13.9 Obtain ATAC program from the following Internet ftp address.

ftp site: `flash.bellcore.com`, *login*: `anonymous`, *password*: `your address`, *directory*: `atac`

Get the `README` file and the archive file "`atac3.3.13.tar.Z`" or the latest version. Read the `README` file and get familiar with the `atac` package.

a. Type up the program `binary.c` shown in Fig. 13.9. Apply `atac` to count its blocks, decisions, c-uses, and p-uses. Verify that there are indeed 1 block, 1 c-use, and 5 p-uses in line 13. You need to write a simple `main.c` program to call the `binary` routine.
b. Apply `atac` to the program in Prob. 13.2 and measure the coverages obtained from your test sets.
c. Apply `atac` to the mutated program in Prob. 3.13 and determine which of your test cases detect the error.