Software risk assessment

by SUSAN A. SHERER and ERIC K. CLEMONS

The Wharton School, University of Pennsylvania
Philadelphia, Pennsylvania

ABSTRACT

This paper presents a framework for software quality that is firmly grounded in the economic significance of software failure. We introduce the concepts of software exposure and software risk—the magnitude of the potential loss due to software failure, and the expected value of this loss, respectively. These, we feel, are far more meaningful measures of software quality than the more traditional expected number of residual errors or mean time between failure, which have been adapted from hardware reliability theory. Our measures can be used by management in software engineering; examples of such use include allocation of test time to modules and the determination of optimal release time for systems.
INTRODUCTION

The past decade has shown an increasing reliance by government, corporations, and individuals on computer software. Evidence suggests that this reliance on software will increase even more rapidly in the future. Many functions previously performed manually are being computerized; few functions, once automated, lose computer support. The increasing computerization of society is creating more dependence on software, particularly for very critical functions such as air traffic control and nuclear power plant control. A growing number of systems are operating in real time, which decreases or eliminates the opportunity for human intervention, further contributing to our reliance on software. Corresponding to the increased dependence on computers has been the development of much more reliable hardware. This improvement in hardware has increased the significance of software as a source of failure.

As reliance on software grows, the importance of developing and ensuring high quality and, in particular, reliable software increases. This requires improved methods for evaluating and controlling quality and reliability in software development. Since software failures have different consequences, any measure of software reliability must include the measurement of the consequence of failure. A measure of software risk—the cost of failure weighted by the probability of failure—is necessary.

Software risk measurement can be used in developing and maintaining high quality software in several ways. First, it may be used to guide software testing. Generally, it is not possible to test a software system until perfection is assured. Trade-offs must be made concerning allocation of test resources: although all portions will be tested, some may be tested more intensively than others. It is reasonable to consider allocating test resources based upon software risk since consequences of failures differ among modules.

Risk measurement of software can be used to determine optimal release times for a software product. The risk of software failure can be weighed against the cost of additional test time to determine the most cost effective time to release the software.

Since the consequences of software failure can be catastrophic, it is reasonable for producers and, perhaps, users to want to have some insurance against their losses. If software risk were quantifiable, insurance could be offered on software products. Just as the insurance industry has grown to meet the demand to support risky ventures, it is reasonable to expect that it should consider encompassing the risk of software failures as this risk becomes a significant part of organizations' and individuals' operations. A means of quantifying this risk is necessary to establish insurance requirements.

This paper begins with a background of the software reliability research and an overview of testing methodologies. Neither traditional reliability measurement techniques nor traditional testing methodologies consider the fact that the implications of software failures vary. Software reliability measurement techniques have been adapted from hardware reliability, where a failure typically has a single consequence. However, since different software failures have different expected consequences, traditional reliability measurement techniques are inappropriate for measuring the true economic risk associated with failure. Traditional testing methods do not consider expected consequences of errors and thus are of only limited use in allocating available resources to the portions of the system with the greatest risk.

The paper introduces and discusses a methodology for assessing software risk that considers both the magnitude and likelihood of loss. Measurement of risk begins with an assessment of external exposure, that is, the magnitude of loss due to invalid actions. The external exposure is mapped onto the system to determine the magnitude of loss due to failures caused by faults in individual modules. Assessment of the likelihood of failure for each module is based upon characteristics of the code and the development process as well as results of test efforts.

BACKGROUND

Software Risk Versus Reliability

Software risk is defined here as the cost of failure weighted by the probability of failure. This definition differs significantly from current definitions of software reliability. The major difference is its inclusion of a measure of the cost of failure. Software reliability models have not considered the cost of failure. They have been adapted from hardware reliability assessment, which models failure as producing a single, known consequence, generally total system failure. However, the consequences of software failure are more varied. Furthermore, the causes of failure differ: hardware may fail after use as components fatigue or wear out; software may fail as new use encounters old errors. We may therefore expect different statistical properties for hardware and software failures. Thus, software risk assessment differs from hardware reliability assessment, and it is not surprising that traditional software methods, grounded in the hardware tradition, should prove less than wholly satisfactory.

There have been several definitions advanced for software reliability. The early software reliability models defined reliability in terms of the number of residual errors in a program. Other models have adapted the traditional hardware
definition of reliability, mean time to failure, in defining soft-
ware reliability as mean time between software failures.9-10
Littlewood defines reliability as a strength of our belief in the
operation of the software without failure for a given time
period.11,14

The major problem with all of these definitions lies in the
assumption that all errors or failures are the same. Whereas
hardware failures typically have the same consequence (i.e.,
the system is not operational), a software error can have a
variety of consequences. There are different types of software
failures. A software failure can be a system crash or a misread
number. If the latter, the consequence of this misread number
can vary. In an air traffic control system, if the hardware fails,
we typically lose sight of all aircraft. If the software fails, we
might lose sight of all aircraft, or we might unknowingly lose
sight of a single aircraft, or we could transpose a digit on the
aircraft identification. The consequence of these different fail-
ure modes are quite different. The point is that software er-
ors and resulting failures have varying consequences, while
hardware failures typically have a single consequence; this
makes hardware models of software failure of limited use-
fulness.

There are other differences between software and hardware
that distinguish software risk assessment from hardware re-
liability measurement. Hardware failures are due to de-
gradation of components as well as design errors. The former
source of failure produces statistically measurable failure pat-
terns. Software components, however, do not degrade as a
result of environmental stress or fatigue. In fact, an older
program is usually more reliable. Software failures are often
due to design and implementation errors that occur as a result
of unanticipated or untested inputs. The correction of soft-
ware errors usually alters the software; it does not just replace
it as in hardware. Finally, software can be copied, retaining
the reliability, whereas hardware cannot.

Software Testing

Complete testing of a software system, conclusively demon-
strating the absence of any errors, is usually impossible.
Therefore, the key issue in software testing has been the selec-
tion of the subset of all possible test cases that has the highest
probability of detecting the most errors.15
Several methodologies are used for accomplishing this task.
These can be classified as:

  White box testing  Black box testing  Random testing

These methodologies are used in various phases of the testing
process. We review the testing process to show that traditional
methods have not adequately addressed the economic con-
sequences of software failure. Since errors can produce con-
sequences of different significance, testing ought to be con-
cerned with the selection of test cases with the greatest
expected loss.

White box, or structural testing, uses the internal structure
of the program to develop test cases. Coverage criteria include
statement coverage, decision coverage, condition coverage,
and combinations of these.15-19

Black box, or functional program testing, is largely data
 driven. The testing process involves partitioning the input
space into equivalence classes. These are sets of input states
that appear to be similar, so that a test of a representative
value in a class should yield results equivalent to a test of any
other value in that class.15 Boundary values of these classes in
particular are usually tested.15,16,19

Random input testing chooses test cases by randomly se-
lecting inputs from the input space, using the same proba-
bilities of selection of input states as occur during operation.20
In many cases, more efficient testing is accomplished if one
recognizes that once an input state has been selected, it does
not have to be repeated. In this case, the failure intensity* must
be divided by a test compression factor, or ratio of
execution time required in operation to execution time re-
quired in test phase, to obtain the corresponding failure fail-
ure intensity to be expected in operation.20

Software testing is a major component of systems develop-
ment, typically accounting for as much as one half of the
development effort.21-22 The testing process generally com-
prises several phases:

  1. Module test  2. Integration test  3. Function test
  4. Systems test  5. Acceptance test

Module, integration, and function testing are typically per-
formed by the systems development group. These phases ver-
ify the code against the design and specification. They
typically consume 45% of the total development effort.23 Sys-
tems and acceptance testing are generally performed by an
independent group, which verifies the system against the
user’s objectives. These phases typically consume only a small
part of the total systems development effort.23

The purpose of module testing is to compare the code to the
module specification. The objective is to show how the mod-
ule contradicts the specification (i.e., to find faults in the
code). Typically, module testing involves a combination of
white box and black box tests. The test manager is faced with
many decisions such as what modules to test, what test data
are necessary, and how to allocate personnel. In many cases,
test resources are constrained. Commonly, test time is lim-
ited. Personnel may also be limited. Thus, the manager must
decide how to allocate test effort to each module so as to
locate as many errors as possible. Typically, this is done on an
ad hoc basis based upon the logic in the code (white box
testing) and the equivalence partitions (black box tests).

During integration testing, the parts of the code are put

* Failure intensity has been defined as failures per unit time.20
ten, the types of test tools used, the cost of generating test cases and the cost of debugging errors.15

Function testing is the comparison of the system to the external specification, a description of the system from the user's point of view. Thus, the system is not tested against the design but against the user specifications. Function testing is typically a black box process. During this test phase, management must decide how to allocate test effort to different data categories so as to uncover as many errors as possible. Typically, time and personnel constrain testing efforts.

The function of the systems test is to compare the system or program to its requirements.15 Its purpose is to assess the system against its original objectives, as opposed to the specifications. The objective of the systems test phase is to produce a reliable software product. During this phase, testing generally proceeds by randomly choosing inputs based upon the user objectives. Software is tested until the deadline for system release is reached or the test team feels that the software is reliable enough for release.19 Software reliability measurement techniques have been developed to determine when desired reliability levels are reached.20 Since they do not consider the consequence of failure, costs and benefits cannot be addressed, and desired reliability cannot adequately be defined or measured.

In all of these types of testing, prior theory never explicitly considers the fact that the errors that we are looking for vary in their consequence. This is a critical factor. Given the constraints that exist during each testing phase, we should be considering the consequence of failure when allocating test efforts. During the module test phase, management should be allocating efforts not only based upon the structure of the code and the equivalence partitions but also considering the different consequences of failure in different modules. During function testing, we ought to be allocating efforts based upon the criticality of the function. During systems test, test effort should be allocated to requirements with the greatest consequences of failure. Optimal release times considering the cost of failure can then be determined. These points are addressed in our research, which is introduced in the following section.

ASSESSMENT OF SOFTWARE RISK

Introduction

We present in this section our extensions to software quality assessment, intended to capture the economic significance of software malfunction.

We define software risk as the expected loss due to failure during a given time period. Risk is measured by the frequency or likelihood of loss (events resulting in loss per unit time) times the magnitude of loss or the level of exposure due to loss (consequences of events).**

In order to develop a measure of software risk that may be used to guide testing, we need to perform several functions. These will be briefly described here and then will be discussed in more detail later in this paper:

1. External exposure identification: What actions, external to the system, can result in losses and what are the consequences of these actions?
2. Structural exposure analysis: What system failures can result in these actions? What is the potential magnitude of loss due to failures caused by faults in each module?
3. Prior predictions of failure likelihood and risk: What is the a priori estimate of the likelihood of failure due to faults in each untested module? What is the resulting estimate of risk?
4. Updating priors using test results: How do we use test results to update failure assessments?

External Exposure Identification

The first step in the measurement of software risk is external exposure identification. This involves an assessment of factors operating in the environment, external to the software, that can contribute to loss. First, we must identify potential actions that can result in loss. Then, we must assess their consequences, that is, the magnitude of loss due to these actions.

Assessment of the external environment involves identification of sources of catastrophe, typically operator actions (or inactions) that can cause disaster by violating norms of behavior. For example, in an air traffic control system, the collision of two airplanes may be caused by an operator who does not take action to change headings when two planes are on a collision course. Our analysis of behavior linked to catastrophic events may reveal items that may have inadvertently been left out of the system requirements. This may yield an additional benefit from this analysis giving us another technique for validating the system design.

The magnitude of loss that may result from inappropriate actions is a function of the environment and the context in which the system operates. The potential loss if an air traffic controller operates inappropriately in situations where planes are on potential collision courses will be much larger if the controller is working during rush hour in a large airport than if he were working on an off hour in a small airport that does not handle wide-bodied aircraft. Generally, there are a large number of environmental factors that must be considered. We will begin by using expected values for the magnitude of loss.

Structural Exposure Analysis

In structural exposure analysis, we map invalid actions, identified in external exposure identification, back to system causes. We try to associate these system causes with potential faults in various modules; we can then identify the magnitude of loss due to faults in various portions of the system.

The magnitude of loss due to system failure is defined as the exposure level. The exposure level is based upon the magnitude of loss due to actions that can result from this failure. Failures are due to faults, or defects, in the system. Our
objective is to assign an exposure level due to faults in the basic interconnected components of the system, the individual modules.

Operator, or environmental, malfunction results from invalid system output. (Note: The absence of expected output may itself also be considered invalid output.) This invalid output is a result of a failure, due to one or more faults in the system. Thus, it is necessary first to relate invalid actions to system failures. For example, failure of an operator to respond to aircraft on a collision course may be due to the system’s failing to display more than one aircraft with conflicting headings.

After identification of potential system failures, we attempt to determine which modules or paths in the system may have faults that can result in these failures. In the above example, we must determine which modules process data related to the display of aircraft heading. To do this, we need to determine how the output resulting from the system malfunction is produced. Invalid output, triggering actions resulting in losses, is produced by invalid processing of data by the system.

To assign exposure levels to modules, we will assume that the fault potential of a module is a function of the type of processing taking place in each module. The functions relating to data with the potential for producing invalid outputs causing loss are identified.

The module’s function at any time, and the associated exposure level, may vary with the way the system is using it; this, in turn, is related to the way the system itself is being used. For example, response to an air traffic control function produced by a module can result in different actions, based upon the purpose of the display at the time of failure. Thus, it will be necessary to evaluate the complex relationship of module functions to system functions.

In sum, the exposure level of a module is based upon the actions that can result from failures due to faults in the module. It depends upon what data is processed by the module and how that data relates to invalid output. It also depends upon how the module processes the data, itself a function of the nature of the use of the module.

Prior Prediction of Failure Likelihood and Risk

The likelihood of a software failure depends upon the number of faults or errors in a program and the probability that a fault or program defect will be encountered in operation, causing a failure.

The number of faults is a function of the product as well as the development process. Most research has concentrated on studying the relationship between characteristics of the final product and the number of faults found. Many researchers have found that the size of a program, measured in terms of the number of executable source statements, has the most effect on the number of faults it contains.\textsuperscript{20,25} Research relating measures of program complexity to the number of faults has been inconclusive.\textsuperscript{20,27} Characteristics of the development process also affect the number of faults. This includes such factors as skill level of the development team, communication among the team members, quality of reviews, and familiarity with application and techniques. However, it is difficult to develop objective measures of these characteristics.

One means of relating the number of faults to both the product and the development process would be to gather historical information concerning the number of faults in modules developed by individual programmers. We will assume that programmer performance, measured in terms of number of errors per line of code, remains fairly stable over time after an initial learning period. This assumption may be relaxed at a later date to account for experience and learning factors. By measuring the number of errors per line of code for each programmer, we gain useful historical information concerning the development process; this information can be used to make prior assessments of the programming product.

The probability that a fault produces a failure depends upon the number of ways in which the module can be used and the frequency with which the module is used. If a module has a large number of paths and input classes, the probability of a specific fault causing a failure on any given run is much less than if a module has only a single path and a single input class. Typically, in operation, certain sections of code are used more often than others, resulting in unequal per fault hazard rates. The operational profile, or set of all possible input states with their associated probabilities of occurrence, will determine which sections of the code will be used most frequently. If the fault is located on a main branch of the code or in a portion of code well traversed, it should have a higher probability of causing a failure than if it is located in a section of code rarely traversed.

The risk for each module is equal to the probability of each type of failure times the cost of that failure. Since we do not know the probability of each type of failure, we will begin by estimating risk for each module as the product of the expected exposure level times the aggregate failure likelihood for that module. The expected exposure level will depend upon the expected use of the system during operation. Failure likelihood depends upon the number of faults and the probability that faults will produce failures.

Updating Priors Using Test Results

As testing proceeds, we gain information concerning the software risk. This may change our initial perceptions, or prior assessments, of the magnitude and location of the risk. Before testing, we assessed the exposure level based upon our prior identification of the types of failures that could occur and of their consequences. We estimated the number of faults and the likelihood of failure based upon a priori perceptions of the development process and characteristics of the code. Examination of failure and debugging information will yield new knowledge about the system.

The frequency and number of failures due to faults in each module may change our initial estimate of the failure likelihood for each module. One means of doing this is to assume a statistical distribution for software reliability growth. As failure data become available, statistical inference procedures may be used to update the parameters of the distribution.\textsuperscript{8,10,11,28} In most cases, these parameters are the number
of faults and the probability that a fault will cause a failure.

A second means of using failure data to update our prior estimates is based upon the observation in many systems that errors tend to cluster. As one example, this phenomenon was observed in IBM’s S/370 operating systems. In one of these operating systems, 47% of the errors found by users were associated with only 4% of the modules within the system. In fact, one of Myers’s testing principles is that “the probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.” Thus, the location of many faults in a single module may change our prior estimate of the number of faults still to be found in that module.

Test results also give us new information about programmer performance that may be used to update our estimates of failure likelihood. Test data on several modules by the same programmer indicating many more faults than initially expected in those modules would cause us to increase our expectations of faults in as yet untested modules by this programmer. We can use this information to update prior estimates of number of faults and, thus, the probability of failure in modules by this programmer.

CONCLUSIONS

We have developed a new measure of software quality, software risk. Measures of software reliability should attempt to consider the varying significance of software failures. Once we can assess the variation in software risk between different portions of a system, we can more cost effectively test our systems.

We have presented a framework for measuring software risk and using the measurement in testing. The next step is to develop the supporting mathematics representing our assumptions about the failure likelihood. This will be needed to develop prior estimates of failure likelihood as well as to update these estimates with test information. Empirical results will then be needed to support the theory.

REFERENCES

The various environments in the modern workplace encompass not only technical and business activities but also structural issues of the department, the company, and the ever changing larger society. In this track, some timely applications and associated problems are presented in detail, and their relationships to organizational and societal concerns are shown, in addition to their taxonomy and interrelationships. A featured session examining the general environment of workplace applications is arranged to introduce the professional and nonprofessional alike to this track. Following are some glimpses into the activities of the other sessions scheduled:

- A plethora of intelligent applications in the office of the future and the connected problems on the human side.
- A well-orchestrated battle to decide the winner among the page description languages.
- CIM, computer integrated manufacturing, which will be the panacea for the problems associated with the factory of the future and the different facets of these stages of integration.
- Effective data analysis and interpretation associated with the mechanical CAE, where various techniques are introduced to handle the specific problems.
- Industry’s attempts to standardize the CAE systems for electrical design applications, thus aiming at improved productivity, data sharing among various diverse systems, and the current status in these efforts.
- The expert systems for reliability in complex designs with stringent requirement constraints.
- The rapid pace of computing in the modern financial world, covering the spectrum from applied AI to technological advances in the marketing field.
- The advance of computers in medical research, pharmaceutical product management, and private practice.