12 Reliability in Computer Systems

12.1 INTRODUCTION

In recent years, the applications of computers have increased quite dramatically ranging from personal use to control space systems. Today's computers are much more complex and powerful than their infant versions. As the computers are made up of both the software and hardware components, the percentage of the total computer cost spent on software has changed quite remarkably from the days of the first generation computers. For example, in 1955 software (i.e., including software maintenance) accounted for 20% of the total computer cost and in 1985, the software cost increased to 90% [1].

For an effective performance of a computer, both its hardware and software must function with considerable reliability. In fact, according to various past studies, the software is many times more likely to fail than hardware. Nonetheless, as computers are becoming more complex and sophisticated, the demand on their reliability has increased exponentially. For example, NASA's Saturn V Launch computer (circa 1964) had a mean time to failure (MTTF) goal of 25,000 h and the SIFT and FTMP avionic computers designed and developed in the late 1970s to control dynamically unstable aircraft were expected to have a MTTF of 1 billion hours [2, 3]. The early history of computer hardware reliability may be traced back to the works of Shannon [4], Hamming [5], Von Neumann [6], and Moore and Shannon [7]. For example, the triple modular redundancy (TMR) scheme was first proposed by Von Neumann [6] to improve system reliability in 1956. Since then, many other people have contributed to the computer hardware reliability and a comprehensive list of publications on the subject is given in Reference 8.

The serious effort on software reliability appeared to have started at Bell Laboratories in 1964 [9]. An evidence of this effort is a histogram of problems per month concerning switching system software. In 1967, Floyd [10] considered approaches for formal validation of software programs and Hudson [11] proposed Markov birth–death models. Since those years, a large number of publications on the subject have appeared [8].

This chapter presents important aspects of computer hardware and software reliability.

12.2 TERMS AND DEFINITIONS

As there are certain terms specifically concerned with computer hardware and software reliability, this section presents some of those terms along with their definitions [8, 12-15].

© 1999 by CRC Press LLC

- **Software error.** This is a clerical, conceptual, or syntactic discrepancy which leads to one or more faults in the software.
- **Fault.** This is an attribute which adversely affects an item's reliability.
- **Covered fault.** This is a fault from which the concerned item can recover automatically.
- **Uncovered fault.** This is a fault from which the concerned item cannot recover automatically.
- **Debugging.** This is the process of eradicating and isolating errors.
- **Fault-tolerant computing.** This is the ability to execute specified algorithms successfully irrespective of computer hardware failures and software errors [16].
- **Software reliability.** This is the probability of a given software operating for a stated time interval, without an error, when used under the designed conditions on the stated machine.
- **Software testing.** This is the process of executing software to determine if the results it produces are correct [15].

12.3 HARDWARE RELIABILITY VS. SOFTWARE RELIABILITY

In order to have a better understanding of the differences between computer hardware and software reliability, Table 12.1 presents comparisons of the important concerned areas [8, 17–19].

12.4 COMPUTER FAILURE CAUSES

Computers fail due to many different causes. The major sources of computer failures are shown in Figure 12.1 [20, 21]. These are human errors, communication network failures, peripheral device failures, environmental and power failures, processor and memory failures, mysterious failures, saturation, and gradual erosion of the data base. The first six of these system failure sources are described below.

Human errors are, in general, the result of operator oversights and mistakes. Often operator errors occur during starting up, running, and shutting down the system. Communication network failures are associated with inter module communication and, usually, most of them are of a transient nature. The use of "vertical parity" logic can help to detect over two-thirds of errors in communication lines. Peripheral device failures are important but they seldom cause a system shutdown. The frequently occurring errors in peripheral devices are transient or intermittent and the usual reason for their occurrence is the electromechanical nature of the devices. The causes, such as electromagnetic interference, air conditioning equip- ment failure, earthquakes, and fires, are responsible for the occurrence of environ- mental failures. Power failures can occur due to factors such as transient fluctuations in voltage or frequency and total power loss from the local utility company.

Usually, processor errors are catastrophic but they occur quite rarely, as there are occasions when the central processor fails to execute instructions properly due

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

Downloaded by [University of Ottawa] at 14:51 15 September 2014

TABLE 12.1
Comparison of Hardware and Software Reliability

No.	Hardware reliability	Software reliability
1	Wears out.	Does not wear out.
2	Many hardware parts fail according to the bathtub hazard rate curve.	Software does not fail according to the bathtub hazard rate curve.
3	A hardware failure is mainly due to physical effects.	A software failure is caused by programming error.
4	The failed system is repaired by performing corrective maintenance.	Corrective maintenance is really redesign.
5	Interfaces are visual.	Interfaces are conceptual.
6	The hardware reliability field is well established, particularly in the area of electronics.	The software reliability field is relatively new.
7	Obtaining good failure data is a problem.	Obtaining good failure data is a problem.
8	Usually redundancy is effective.	Redundancy may not be effective.
9	Potential for monetary savings.	Potential for monetary savings.
10	Preventive maintenance is performed to inhibit failures.	Preventive maintenance has no meaning in software.
11	It is possible to repair hardware by using spare modules.	It is impossible to repair software failures by using spare modules.
12	Hardware reliability has well-established mathematical concepts and theory.	Software reliability still lacks well-established mathematical concepts and theory.
13	Has a hazard function.	Has an error rate function.
14	Mean time to repair has significance.	Mean time to repair does not have any significance.
15	Reliability is time-related with failures occurring as a function of operational/storage time.	Reliability is not time-related and failures occur when a program step/path containing the fault is executed.

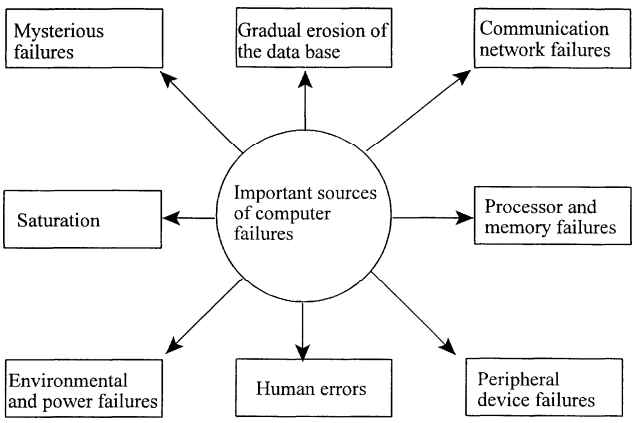


FIGURE 12.1 Major sources of computer failures.

© 1999 by CRC Press LLC

to a "dropped bit". Nowadays, memory parity errors are very rare because of an impressive improvement in hardware reliability and also they are not necessarily fatal. Mysterious failures occur unexpectedly and thus in real-time systems such failures are never properly categorized. For example, when a normally functioning system stops operating suddenly without indicating any problem (i.e., software, hardware, etc.), the malfunction is called the mysterious failure.

12.5 SOFTWARE LIFE CYCLE PHASES AND ASSOCIATED ERROR SOURCES

A software life cycle may be divided into many different phases: requirements definition and analysis, preliminary design, detailed design, code and unit testing, integration and system testing, acceptance testing, maintenance and operation, and retirement. Within each of these phases, there are various sources of error, thus requiring careful consideration. All of these life cycle phases are described below [12].

The requirements definition and analysis phase defines and documents the engineering requirements for the software to be developed and its end product is a software requirement specification that allows the development organization to proceed with the design. Some of the sources that may generate errors in this phase include incomplete requirements, conflicting requirements, inadequately described user needs, and nonfeasible requirements.

In the preliminary design phase, the software product is decomposed into a hierarchal structure consisting of computer software configuration items. In turn, these configuration items are decomposed further into computer software components and modules. During the detailed design phase, the computer software component design specifications are progressively described in more detail until the identification of a set of primitive (i.e., non-decomposable) computer software units. Some of the sources that may induce errors in the detailed design phase are inconsistent data definitions, input data range errors, inadequate validity checking, unsatisfactory error recovery, and wrong error analysis for algorithms.

In the code and unit testing phase, the programmer translates the detailed software design into the programming language and the unit testing is performed to verify unit functionality. As coding is a fault-prone process, a number of standard bugs that may occur during this phase include missing code, incomplete initialization or resetting of variables and parameters, unreachable code, improper logic for branch conditions and loops, out-of-range calculations and infinite loops, and improper or incomplete validity checks.

At the completion of the unit tests, the integration testing starts and the separately tested units are assembled into logical combinations and tested again to demonstrate that the assembly adequately satisfies product requirements. The process is repeated by combining larger and larger classes of units until the full integration of all units of a computer software configuration item. The purpose of the system testing is to demonstrate that the software and hardware work together to satisfy all system functions as well as that the end product is ready for release to production. The

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

errors/faults introduced in the test phase significantly influence the test program's capability to uncover faults in the software and those errors/faults include test plans/procedures that have wrongly interpreted software requirements and errors/faults in code written for the test program.

In the acceptance testing phase, the test team determines if the product under consideration meets its original requirements. The acceptance test plan is developed and the phase terminates when all the tests in the acceptance plan are successfully executed.

In the maintenance and operation phase, primarily the attention is focused on rectifying errors that appear in the exercise of the software or on fine-tuning the software to improve its performance.

In the retirement phase, the user may decide not to use the software any more and discard it altogether. The primary reason is that the software may have become difficult to maintain because of recurrent changes.

12.6 SOFTWARE RELIABILITY IMPROVEMENT METHODS

There are a number of methods to improve reliability of software products. The classifications of these methods include reliable software design, fault-tolerant design, formal methods, and testing.

12.6.1 Reliable Software Design Methods

Under this classification, there are a number of techniques that can help programmers to systematically derive the software design from its detailed specification [22]. These include structured programming, top-down programming, and data structure design.

The clarity of the design is the direct aim of the *structured programming*. More specifically, the structured programming calls for easy to understand design and a need to develop a program structure in which the vital software elements and their inter-relationships are easily identifiable. Some of the easy to use rules associated with structure programming include restricted use of GO TO statements in modules, ensuring that each module has only one entry and one exit, using a language and compiler with structured programming capabilities for the implementation purpose, and ensuring that all statements, including subroutine calls, are commented [23].

There are many advantages associated with the structured programming: increase in the productivity of programmers with respect to instructions coded per man-hour, a useful tool in localizing an error, maximum amount of code can be reused in redesign work, and an effective tool in understanding the program design by the designer and others. The *top-down programming* or design is also sometimes referred to as hierarchic decomposition because of the cascading nature of the process [24]. Nonetheless, the top-down design is a decomposition process concerned with directing attention to the program control structure or to the program flow of control. Furthermore, the top-down design is also concerned with the production of code at a later stage.

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

The top-down programming begins with a module representing the program in its entirety and it decomposes the module into subroutines. In turn, each subroutine is broken down further and this process continues until all the broken down elements are easy to comprehend and work with. The program structure is analogous to a tree; the trunk of the tree represents the module containing the entire program and its leaves represent the computer language statements. Some of the benefits of the top-down structured programming are reduction in the cost of testing, better quality software, software in a more readable form, lower cost of software maintenance, increase in confidence in software, etc.

The *data structure design* is inclined toward the development of a detailed program design and is also referred to as Jackson Structured Programming. The structure of the input files drive the methodology and its resulting design and from these files software is read and by the structure of the output files it is generated. According to this approach, an effective software design reproduces the data structure in the input/output files. Currently, this is the most systematically available design method and is simple, teachable, consistent (e.g., two totally different programmers will end up writing the same program), and anti-inspirational. The downside is that the method is not applicable to scientifically based programs because their contents are essentially algorithmic (i.e., functional).

12.6.2 Fault-Tolerant Software Design Methods

It would never be possible to produce error-free software, irrespective of the degree of testing or other measures [25, 26]. Under such circumstances, one approach to improve software reliability is to create a fault-tolerant architecture to recover from software failures. The expensiveness of the approach usually limits applications to high-risk areas, for example, embedded software for controlling satellites. Three methods of fault-tolerant software design are recovery-block design, N-version programming, and consensus recovery block. The *recovery-block design* is concerned with developing versions of software under the same set of requirements on the assumption that the pieces of software being developed are independent, and thus the likelihood of their failing simultaneously is negligible. The versions are ordered from the most reliable to the least reliable. The most reliable software is executed first and after the acceptance test run, if its output is rejected then a recovery block is entered. The recovery is conducted in the following three steps:

- Recover input.
- Execute most reliable software and restore input.
- Submit the output to the same acceptance test as before. In the event of rejection, the next recovery block is entered and the process continues.

In the case of *N-version programming*, many versions of the software are developed independently, N programs are executed in parallel or simultaneously. Subsequently the resulting outputs are compared and when at least programs have the same output, it is taken as the correct result and the process continues. This approach is not suitable where multiple correct outputs can be generated and also it discrim- inates correct results or solutions affected by rounding errors.

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

The *consensus recovery block method* combines attributes of the preceding two approaches; thus, it attempts to discard the weaknesses of those two methods. Nonetheless, this approach requires the development of N-versions of a software and of an acceptance test voting procedure. The reliability factor is used to rank the different versions of the software after execution of all software versions. The resulting outputs are submitted to the voting mechanism. If no agreement is reached, the order of reliability is used in submitting each output successively to the acceptance test. As soon as one of the resulting outputs passes the test, the process terminates and the software under consideration continues with its operation. All in all, it may be said that this method is more reliable than recovery-block design and N-version programming approaches.

12.6.3 Formal Methods

These include formal specification approaches and formal verification [12, 27]. The formal specification approaches consist of a detailed specification of the software requirements using a formal language. More specifically, the nature language of traditional programming is not used, but rather a "formal" substitute having exactly defined semantics and syntax. A typical example of such a language is mathematics. Nonetheless, the formal specification approaches help to enforce clarity in requirement definitions and leave little room for ambiguity and misinterpretation, which is usually a concern when using a natural language.

The formal verification makes use of inference on the formal specification in order to demonstrate that the software meets its objectives [28]. All in all, it may be said that the formal approaches are not properly understood by the involved professionals because of their complexity and need for considerable training.

12.6.4 Testing

This may simply be described as the process of executing a software program to uncover errors. There are many different types of software testing including module testing, top-down testing, bottom-up testing, and sandwich testing [8, 29–31]. Each of these testing methods is discussed below.

- **Module Testing**. This is concerned with testing each module usually in isolation from the rest of the system, but subject to the environments to be experienced in the system. Usually, a general module test program is developed instead of writing an entirely new program for each module to be tested. Because it is easy to correct uncovered errors and more grave consequences of errors can be discovered at a later stage, it is advisable to conduct module testing thoroughly [32].
- **Top-Down Testing.** This is concerned with integrating and testing the software program from the top end to the bottom end and in the program structure, the top module is the only module that is unit tested in isolation. At the conclusion of the top module testing, this module calls the remaining modules one by one to merge with it. In turn, each combination is tested and the process continues until the completion of combining and

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

testing of all the modules [31]. The advantages of the top-down testing are efficiency in locating problem areas at the higher program levels, easy representation of test cases after adding the input/output functions, and early demonstrations because of the early skeletal program. Similarly, some of the drawbacks of the top-down testing are that stub modules must be written, stub modules may be complex, test conditions could be impossible or difficult to create, and the representation of test cases in stubs can be difficult prior to adding the input/output functions [12].

- **Bottom-Up Testing**. This is concerned with integrating and testing the software program from the bottom end to the top end and the terminal modules are module (unit) tested in isolation. These modules do not call any other modules and at the conclusion of the terminal module testing, the modules that directly call these tested modules are the ones in line for testing. The modules are not tested in isolation; in fact, the testing is conducted together with the earlier tested lower level modules. The process continues until reaching the top of the program. Some of the benefits of the bottom-up testing are ease in creating test conditions, efficiency in locating problem areas at the lower levels of the program, and ease in observating test results. Similarly, the bottom-up testing drawbacks include: driver modules must be produced, and the program does not exist as an entity until the adding of the final module [12].
- **Sandwich Testing.** This method is the result of combining the top-down testing and bottom-up testing methods. Obviously, the idea behind developing this approach was to extract benefits of both top-down testing and bottom-up testing and eliminate some of their drawbacks. As both top-down testing and bottom-up testing are performed simultaneously, the program is integrated from both sides (i.e., top and bottom). Even though the resulting meeting point from both sides depends upon the program being tested, it can be predetermined by reviewing the structure of the program.

12.7 SOFTWARE RELIABILITY ASSESSMENT METHODS

There are many qualitative and quantitative software reliability assessment methods available in the published literature. Such methods may be grouped into three distinct classifications [12]:

1. Analytical approaches
2. Software metrics
3. Software reliability models

12.7.1 Analytical Approaches

Two important techniques belonging to the analytical approaches are failure mode and effect analysis (FMEA) and fault tree analysis (FTA). Both of these methods are widely used to assess hardware reliability and they can equally be applied to

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

assess reliability of software. FMEA and FTA are described in detail in Chapters 6 and 7, respectively.

12.7.2 Software Metrics

These may be described as quantitative indicators of the degree to which a software item or process possesses a specified attribute. Often, the application of the software metrics is made to determine the status of a trend in a software development activity as well as to determine risk of going from one phase to another. Two software measures are described below.

Defect Density Measure

This is an important measure and can be applied to determine the reliability growth during the design phase. The measure requires establishing defect severity categories and possesses some ambiguity, since its low value may mean either an inadequate review process or a good product. Nonetheless, the cumulative defect ratio for design is expressed by

$$\theta_{cd} = \sum_{i=1}^{\alpha} N_i / L \tag{12.1}$$

where

- θ_{cd} is the cumulative defect ratio for design.
- α is the total number of reviews.
- N_i is the total number of unique defects, at or above a given severity level, discovered in the ith design review.
- L is the number of source lines of design statement expressed in thousands, in the design phase.

In the event of having estimates of defect density greater than the ones for comparable projects, review the development process to determine if poor training/practices are responsible or if the requirements are ambiguous or incomplete. Under such circumstance, it may be the correct course to delay development until such time the corrective measures can be taken. In contrast, if estimates of defect density are less than for comparable projects, review the methodology and the review process itself. If the assessed review process is considered to be satisfactory, it is quite reasonable to conclude that the development phases are generating low-defect software products.

Code and Unit Test Phase Measure

For this phase, another form of the defect density measure is more appropriate and again this form requires the establishment of defect severity classifications. Thus, the cumulative defect ratio for code is expressed by

© 1999 by CRC Press LLC

Downloaded by [University of Ottawa] at 14:51 15 September 2014

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/767135020062006030