5

Software Reliability

5.1 INTRODUCTION

Computers are finding an ever-increasing number of applications. The expenditure on computer software is increasing faster than on associated hardware. One of the estimates indicates [1] that the annual expenditure of the U.S. Air Force, for example, on computer hardware is $400 million and the corresponding expenditure for software is estimated at $1500 million per year. This ratio of four to one is predicted to rise to nine to one [5, 23]. With expenditures of this magnitude, it is natural that attention should be directed to the proper development of software for computer applications. One area on which considerable emphasis has been placed in recent years is software reliability. This has come primarily with the advent of large and complex hardware-software systems and the use of computers as the heart of real-time applications to control vital and critical functions. The undetected errors can cause system failures with catastrophic results and at the same time the size and complexity has increased making the process of debugging more difficult. Most of the work in the area of software reliability can be divided into the following three categories:

1. Writing correct programs to begin with.
2. Testing the programs to take out the bugs.
3. Modeling of software in an attempt to predict its reliability and possibly study the impact of related parameters.

These three areas are discussed in this chapter. It should be pointed out that software reliability is in no way as highly developed as the discipline of hardware reliability. Several useful concepts have, however, emerged, and considerable work is still under progress.

5.2 HARDWARE AND SOFTWARE

The discipline of hardware reliability is considerably older than that of software. It is, therefore, natural to make a comparison between the two in an effort to apply the large body of knowledge of reliability engineering to

software assurance. Several attempts [8, 13] have been made in this direction. It appears that much can be learned from established reliability engineering in organization and control procedures but that there are significant differences when it comes to failure mechanisms. A hardware component is assumed to have failed if its characteristics change beyond the design values either by drift or catastrophic failures. A piece of software, however, does not fail. If a program does not do what it is supposed to do, it is because an error is present. The error has been there and when the segment of the program containing the error is energized, the error becomes manifest. This encountering of error may or may not cause a system to fail. Whereas the hardware undergoes a change at the instant of failure, the software is really the same as it was before the error was discovered.

The hardware reliability of a system can be improved by using two identical components in a redundant manner. Two identical softwares, however, will be of little use in increasing the reliability since the same error will be exercised in both at the same time.

There is an important difference between hardware and software in regard to the relationship between testing and reliability. If the software could be tested for every conceivable input, then theoretically it should never cause system failure. The hardware, on the other hand, could fail even after having been tested in the most exhaustive manner.

A question that may be asked is, "Can the failure behavior of software be regarded as random?" A program basically maps the elements of input space into corresponding elements of output space [11]. A certain subset of the input space would produce incorrect output. If we knew the output behavior for every conceivable input and could predict the future inputs, then we could predict the failures in a deterministic fashion. The properties of a large piece of software, however, may never be known completely, since it is almost impossible to test software for every conceivable input. The input to the software is also random. With the uncertainty associated with both the input and the software, randomness can be justified for the occurrences of errors.

5.3 SOFTWARE RELIABILITY MODELS

Reliability models can be used to predict the reliability when the software is put into operational use. Several models have been proposed [21] in the literature, and a few are described here. The software reliability models use the information of the number of errors debugged during the development of a software program. This information is used to characterize the model parameters that can then be used to predict the number of failures or some other measure of reliability in the future. The software reliability can be defined as the probability of a given software operating for a specified time

period, without a software error, when used within the design limits on the appropriate machine.

5.3.1 Shooman Model

The model proposed by Shooman [17, 18] is based on the following assumptions:

1. The total number of machine language instructions in the software program is constant.
2. The number of errors at the start of integration testing is constant and decreases directly as errors are corrected. No new errors are introduced during the process of testing.
3. The difference between the errors initially present and the cumulative errors corrected represents the residual errors.
4. The failure rate is proportional to the number of residual errors.

Based on these assumptions [17],

$$e_r(x) = e(0) - e_c(x) \tag{5.1}$$

where x = debugging time since the start of system integration
$e(0)$ = errors present at $x = 0$, normalized by the total number of machine language instructions
$= E_0/I$
E_0 = number of initial errors
I = total number of machine language instructions
$e_c(x)$ = cumulative number of errors corrected by x, normalized by I
$e_r(x)$ = residual errors at x, normalized by I

Assuming failure rate to be proportional to residual errors (assumption 4),

$$\lambda_s(t) = K_s e_r(x) \tag{5.2}$$

where t = operating time of the system
K_s = constant of proportionality
$\lambda_s(t)$ = failure rate at time t

Knowing the failure rate from (5.2), the expression for reliability or survivor function [20] is

$$R(t) = \exp\left[-\int_0^t \lambda_s(x)\,dx\right] = \exp\left[-\int_0^t K_s e_r(x)\,dx\right]. \tag{5.3}$$

Since the hazard rate is assumed independent of t in this model, this assumption amounts to a constant failure rate, and therefore,

$$\text{MTTF} = \frac{1}{\lambda_s(t)} = \frac{1}{K_s e_r(x)} \tag{5.4}$$

Estimation of Model Parameters. By substituting for $e_r(x)$ from (5.1) into (5.4), the expression for MTTF can be written as follows:

$$\text{MTTF} = \frac{1}{K_s[e(0) - e_c(x)]} = \frac{1}{K_s[E_0/I - e_c(x)]} \tag{5.5}$$

There are two unknowns in (5.5), K_s and E_0. These parameters can be estimated using the moment matching method [20]. Considering two debugging periods x_1 and x_2 such that $x_1 < x_2$,

$$\frac{T_1}{n_1} = \frac{1}{K_s[e(0) - e_c(x_1)]} \tag{5.6}$$

and

$$\frac{T_2}{n_2} = \frac{1}{K_s[e(0) - e_c(x_2)]} \tag{5.7}$$

where T_1, T_2 = system operating times corresponding to x_1 and x_2, respectively
n_1, n_2 = number of software errors during x_1 and x_2, respectively

From (5.6) and (5.7)

$$E_0 = \frac{I[\gamma e_c(x_1) - e_c(x_2)]}{\gamma - 1} \tag{5.8}$$

where $\gamma = \dfrac{T_1}{T_2} \cdot \dfrac{n_2}{n_1} = \dfrac{\text{MTTF}_1}{\text{MTTF}_2}$

MTTF_i = the mean time to software failures corresponding to debugging time x_i.
$= \dfrac{T_i}{n_i}$

By substituting for E_0 from (5.8) into (5.6),

$$K_s = \frac{n_1}{T_1[E_0/I - e_c(x_1)]} \tag{5.9}$$

An alternative method for estimation of E_0 and K is by using maximum likelihood estimates and is discussed in reference 17.

5.3.2 *The Markov Model*

This model [19, 22] assumes the system to go through a sequence of "up" and "down" states. The system state is termed "up" if the first error since the start of integration and testing has not yet occurred or if the system has been restored after an error and the next error has yet to be encountered. The down state implies that an error has occurred and has not been corrected. The state transition diagram of this model is shown in Figure 5.1, in which the following notation is used:

1. State $(n-k)$ indicates that kth bug has been corrected and that $(k+1)$th error is yet to occur. This is the up state following the down state due to the kth bug.
2. State $(m-k)$ is entered when the $(k+1)$th bug is discovered. This is the down state due to $(k+1)$th error.
3. λ_k is the error occurrence rate when the system is in state $(n-k)$.
4. μ_k is the error correction rate when the system is in down state $(m-k)$.
5. $P_j(t)$ is the probability of the system being in state j at time t.

The state differential equations for this system can be easily formulated using known methods [20].

$$\dot{P}_{n-k}(t) = -\lambda_k P_{n-k}(t) + \mu_{k-1} P_{m-k+1}(t) \tag{5.10}$$

$$\dot{P}_{m-k}(t) = -\mu_k P_{m-k}(t) + \lambda_k P_{n-k}(t) \tag{5.11}$$

The initial conditions are

$$P_{m-k}(0) = 0 \qquad k = 1, 2, 3 \ldots \tag{5.12}$$

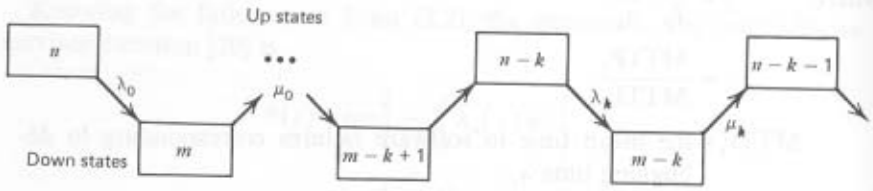

Figure 5.1 The Markov model.

and

$$P_n(0) = 1 \tag{5.13}$$

A restrictive solution of (5.10) and (5.11) assuming constant $\lambda_k = \lambda$ and constant $\mu_k = \mu$ is derived in [22]. This constraint on λ_k and μ_k is, however, easily seen to be unrealistic. A more general solution can be obtained using numerical techniques like Euler's and Runge-Kutta methods for integration. Once the state probabilities have been obtained, unavailability is calculated as [19, 22]

$$U(t) = \sum_{k=0}^{k_{\max}} P_{m-k}(t) \tag{5.14}$$

The probabilities will depend on the choice of $k_{\max}$. By choosing $k_{\max}$ large enough, $U(t)$ can be made close to the true value of $U(t)$.

5.3.3 *Jelinski-Moranda Model*

This model [14, 15, 21] like the Shooman model assumes an exponential probability density function for software errors. The hazard rate is assumed to be proportional to the number of remaining errors, that is,

$$\lambda_{JM}(x_i) = K_{JM}[E_0 - (i-1)] \tag{5.15}$$

where K_{JM} = constant of proportionality
x_i = time between the ith and $(i-1)$st errors discovered

The reliability function and the mean time to failure can be obtained [14] from (5.15):

$$R(t_i) = \exp[-K_{JM}(E_0 - i + 1)t_i] \tag{5.16}$$

and

$$\text{MTTF} = \int_0^\infty R(t_i)\,dt_i = \left| \frac{-1}{K_{JM}(E_0 - i + 1)} \exp[-K_{JM}(E_0 - i + 1)t_i] \right|_0^\infty$$

$$= \frac{1}{K_{JM}[E_0 - i + 1]} \tag{5.17}$$

5.3.4 *Schick Wolverton Model*

The Schick Wolverton model [24] assumes the hazard rate proportional to the number of remaining errors and the debugging time:

$$\lambda_{sw}(t_i) = K_{sw}[E_0 - (i-1)]x_i \tag{5.18}$$

where x_i = time interval between the $(i-1)$st and the ith error.

The reliability function is

$$R(t_i) = \exp\left[\int_0^{t_i} \lambda_{sw}(x)\,dx\right]$$

$$= \exp\frac{-K_{sw}(E_0 - i + 1)t_i^2}{2} \tag{5.19}$$

$$\text{MTTF} = \int_0^{\infty} R(t_i)\,dt$$

$$= \int_0^{\infty} \exp\left[-K_{sw}(E_0 - i + 1)\,t_i^2/2\right] dt_i$$

$$= \left[\frac{\pi}{2\,K_{sw}(E_0 - i + 1)}\right]^{1/2} \tag{5.20}$$

It could possibly be argued both for and against having the hazard rate proportional to debugging time. Probably the only way to judge the suitability of this model is by fitting it to the experimental data.

5.4 MODEL VALIDATION

Four models have been described in this chapter and several more have been proposed in the literature [21]. In addition to the models described in reference 21, Bayesian models have also been proposed [10]. The true worth of a model can be measured by its ability to predict. Most of the discussion on the relative worth of the models is generally based on intuition and logical consistency. Because of the scarcity of data on software errors and lack of consistency in the available data, only a few attempts have been made for the experimental validation of these models. One such attempt has been reported in the literature [21], wherein a comparative study of the four models described in this chapter and five more models is described.

The error data used by Sukert [21] came from Software Problem Reports (SPR's) during the software development of a large command and control system. The software was written in Jovial J4 code and consisted of 249 routines with a total of 115,000 lines. Although some internal tools such as static code analyzer were used by the contractor for software development, no techniques like structured programming were used.

The data was restructured so that each entry corresponded to a single error and to delete entries due to nonsoftware errors. The data was then sorted according to the date of opening an SPR so as to provide a sequential time frame suitable for input to the models. The data on CPU

time was not available from this project. A day was considered as the basic unit of debugging interval length.

Because of the unavailability of CPU data, the Shooman model could not be used. The other three models along with some modifications of these and several other models were compared and the following conclusions were drawn [21].

1. The Jelinski-Moranda and Schick-Wolverton models consistently gave higher predictions for the number of remaining errors than the actual number, that is, the prediction is conservative or pessimistic with these models.
2. For small software projects or where the testing phase is short, Jelinski-Moranda and Schick-Wolverton models appear to give a reasonable prediction for the number of remaining errors.
3. Of all the models studied, Schick-Wolverton, or a modified version of Jelinski-Moranda models appear to give the best prediction for the remaining errors for large projects or projects with a long-testing phase.

It should be remembered that even though this comparative study has produced some useful results, many more studies of this kind are needed.

5.5 SOFTWARE RELIABILITY ASSURANCE AND IMPROVEMENT

Modeling is only one aspect of software reliability and is intended to predict the number of bugs remaining in the system by using the statistical information on discovering and removing the errors. There are, however, two equally and perhaps more important areas of software reliability. These additional areas can be described as (a) designing for reliability, and (b) testing for reliability assurance. These two topics are discussed in this section.

5.5.1 *Designing for Reliability*

Probably the best way to have reliable software is to minimize the number and severity of bugs while a software package is being developed. There does not appear to be any proven best way of producing reliable software. There is as yet no theoretical framework for techniques for turning out error-free software. However, there appears to be an emerging consensus that certain program structuring and management techniques are conducive to developing reliable software. These techniques are usually referred to as structured programing and several techniques related to it.

Structured Programing. Several definitions of structured programing are floating around. A more generally accepted definition of this approach appears to be coding that avoids use of GO TO statements and program design that is TOP DOWN and modular. These three features appear to enhance program reliability, readability, and maintainability.

Top Down Programing. There are basically two ways of program design, bottom up and top down. The classical way of writing large programs is bottom up. In this approach, the program manager views the project as a whole, determines the system objectives, and then specifies the components needed for the software. The interfaces are specified and the component softwares are allocated to individual programers for development. Each programer is responsible for testing his subassembly or module before it goes into integration. The modules are integrated level by level by the most capable member of the group whose modules are being integrated. This manner of software development is similar to the one used for hardware development.

It appears, however, that an alternative approach of software development in the top down manner gives more reliable software [2]. Here the chief programer programs instead of providing supervision alone. The core of the system is written first assuming dummy subassembly at the next level. These subassemblies are developed next in likewise manner. In comparing these two approaches, an analogy with the chief surgeon is often drawn. The top down approach is like the best surgeon doing the most important or fundamental surgery himself and coordinating the less essential work performed by others [7].

GO TO Free Coding. Dijkstra published a note in 1968 in CACM [6] entitled "Go To statement considered harmful." The title of this communication seems to have had a wide-ranging effect on contemporary programing techniques. The GO TO statement does not create errors by itself. It is the transfer of control that can create meshing of the flow of logic so that the code can become difficult to read. The avoidance of GO TO statements, on the other hand, creates more transparent and readable code. The GO TO free programs are also more straightforward to prove.

Now if it were conceded that GO TO statements should not be allowed in programing, what is the alternative? It has been shown [4] that any flow chart can be constructed using only the single entry exit structures shown in Figure 5.2. These three control structures can be used to write programs that will be free of GO TO statements and in which the program text will correspond more closely to the program execution [3]. This is best illustrated using an example. Figure 5.3 is an example of a program written using GO TO statements and the same program written using the control structures of Figure 5.2 is shown in Figure 5.4. It can be appreciated that whereas the code in Figure 5.3 jumps around the page, the one in Figure

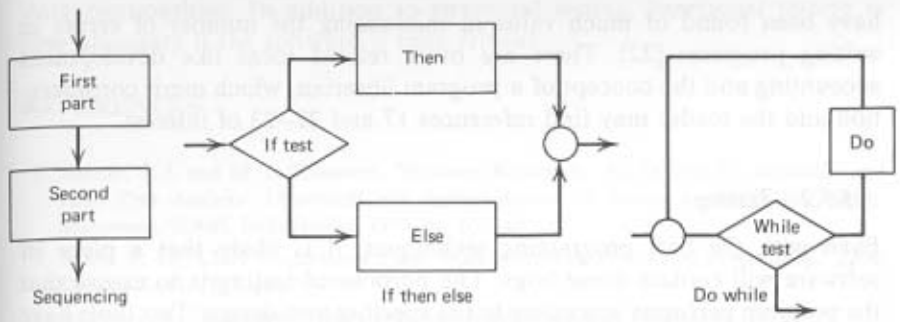

Figure 5.2 Single-entry single-exit logic structures.

```
      A
      IF B   GO TO 10
      D
      GO TO 30
10    E
      GO TO 30
30    G
```

Figure 5.3 Program containing GO TO statements.

```
A
IF B   THEN   E
ELSE D
G
```

Figure 5.4 Program of Figure 5.3 without GO TO statements.

5.4 follows a sequencing process. Imagine a large piece of software written in the manner of Figure 5.3; it could be hard to follow and readily understand, whereas the code written in the manner of Figure 5.4 is more transparent. Such a code is not only readily understood but the programer is less likely to make errors.

Modular Programs. Programers normally break down a complex software development task into separate modules. A module is used by many other modules. This also, however, increases the potential for misunderstanding and errors. It appears that this source of errors can be minimized if every module is entered only at the top and left at the bottom.

Related Ideas. The techniques of structured programing and the concepts of modularity and top down flow have been described. These techniques

have been found of much value in minimizing the number of errors in writing programs [22]. There are other related ideas like development accounting and the concept of a program librarian, which merit consideration and the reader may find references 17 and 22–23 of interest.

5.5.2 *Testing*

Even with the best programing techniques, it is likely that a piece of software will contain some bugs. The purpose of testing is to assure that the program performs according to the specification design. Test tools have been developed to assist in assessing this assurance in computer programs. These tools basically provide some numerical measure of the thoroughness with which the testing was conducted. Several such tools are available and a comparative investigation into the effectiveness of these tools is reported in reference 16.

The test tools consist of the following basic modules: (a) instrumentation module, (b) analyzer module.

The source program of the module under test is first submitted to the instrumentation module, which inserts additional statements into the module. These additional statements are called sensors and counters [16] and the process of adding these statements is called instrumentation. The functional intent of the original code must remain unchanged during the process of instrumentation, that is, the sensor and counter statements must not change the functional objectives of the program.

The instrumented package is compiled in the usual manner and the object package is executed with its test data, which results in an instrumentation data file in addition to the normal output. This instrumentation data file and the instrumented source file are then submitted to the analyzer module, which produces a report indicating the behavior of the module during execution. The following type of information is contained in such a report:

1. The number of times each statement has been executed.
2. At each branch point, how many times a particular path has been taken.
3. Time for executing each statement.

This information is useful in checking the structure of the code. It provides confidence in the logic and code of the program by ensuring that each statement and each branch path has been executed at least once. It is also possible to ensure that each subroutine has been called once. As can be inferred from the description of the testing process, these test tools can be very useful in discovering and reducing sequencing and control errors which account for approximately 20 percent of the total number of errors [16]. These structural analysis tools, however, do not test the timing and data relationships. In addition to structural testing, functional testing is also necessary if the software is time critical.

REFERENCES

1. Amster, S. J. and M. L. Shooman, "Software Reliability: An Overview," *Reliability and Fault Tree Analysis: Theoretical and Applied Aspects of System Reliability and Safety Assessment*, SIAM, Philadelphia, 1975, pp. 655–685.
2. Baker, F. T., "Chief Programmer Team Management of Production Programing," *IBM Systems J.*, **11**, 1972, pp. 56–73.
3. Benson, J. P., "Structured Programing Techniques," see Ref. 13, pp. 143–147.
4. Bohm, C. and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Commun. ACM*, **9**, 366–371 (1968).
5. Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," *Datamation Mag.* **19** (5), 48–59 (May 1973).
6. Dijkstra, E. W., "GO TO Statement Considered Harmful," *Commun. ACM*, **11**, 147–148 (1968).
7. Flynn, R. J., "Design of Computer Software," *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, IEEE, New York, 1975, pp. 476–479.
8. Hecht, H., "Can Software Benefit from Hardware Experience?" *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, IEEE, New York, 1975, pp. 480–484.
9. Management Overview, IBM Installation Productivity Programs Dept., Bethesda, MD (1973).
10. Littlewood, B. and J. W. Verrall, "A Bayesian Reliability Growth Model for Computer Software," see Ref. 13, pp. 70–77.
11. Littlewood, B., "How to Measure Software Reliability and How Not to," *IEEE Trans. Reliability*, **28**, 103–110 (1979).
12. Mills, H. D., "On the Development of Large Reliable Programs," see Ref. 13, pp. 155–159.
13. MacWilliams, W. H. "Reliability of Large Real-Time Control Software Systems," Records 1973 IEEE Symposium on Computer Software Reliability, IEEE Catalog No. 73 CH 0741-9CSR, 1973, pp. 1–6.
14. Moranda, P. B. and J. Jelinski, "Software Reliability Research," In: *Statistical Computer Performance Evaluation*, Edited by Walter Freiberger, Academic, New York, 1972.
15. Moranda, P. L. and J. Jelinski, "Final Report on Software Reliability Study," McDonnell Douglas Astronautic Company, MDC Report No. 63921, Dec. 1972.
16. Reifer, D. J. and R. L. Ettenger, "Test Tools: Are They a Cure-All?" *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, IEEE, New York, 1975.
17. Shooman, M. L., "Operational Testing and Software Reliability Estimation during Program Development," *1973 IEEE Symposium on Computer Software Reliability*, IEEE, New York, 1973, pp. 51–57.
18. Shooman, M. L., "Software Reliability: Measurement and Models," *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, IEEE, New York, 1975.
19. Shooman, M. L. and A. K. Trivedi, "A Many-State Markov Model for Computer Software Performance Parameters," *IEEE Trans. Reliab.*, **R-25**, 66–68, 1976.
20. Singh, C. and R. Billinton, *System Reliability Modelling and Evaluation*, Hutchinson, London, 1977.
21. Sukert, A. N., "An Investigation of Software Reliability Models," *Proceedings of the 1977 Annual Reliability and Maintainability Symposium*, IEEE, New York, 1977.

22. Trivedi, A. K. and M. L. Shooman, "A Many State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters," *Proceedings International Conference on Reliable Software*, April 21–23, 1975, Los Angeles, IEEE Cat. No. 75 CH0940-7CSR, New York.

23. USAF Report, Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's, (CCIP-85), Vol. I, SAMSO/XRS-71, April 1972.

24. Wolverton, R. W. and G. J. Schick, "Assessment of Software Reliability," TRW Systems Group, Report No. TRW-SS-72-04, Sept. 1972.