Software Reliability

H. Kopetz

Software Reliability

H. Kopetz

Technical University of Berlin

M

Authorised English language edition, with revisions, of
Software-Zuverlässigkeit, first published 1976 by Carl Hanser Verlag,
Munich and Vienna

First published 1979 in the United Kingdom by
THE MACMILLAN PRESS LTD
London and Basingstoke
Associated companies in Delhi Dublin
Hong Kong Johannesburg Lagos Melbourne
New York Singapore and Tokyo

Typeset in 10/12 Press Roman by
Styleset Limited · Salisbury · Wiltshire
and printed in Great Britain by
Unwin Brothers Limited
The Gresham Press
Old Woking, Surrey

British Library Cataloguing in Publication Data

Kopetz, H
Software reliability. – (Macmillan computer science series).
1. Computer programs – Reliability
I. Title
001.6'425 QA76.6

ISBN 0–333–23372–7
ISBN 0–333–23373–5 Pbk

Preface

This book is intended for the student of computing and the practising computer professional who is concerned about the unreliability of computer systems. It is the author's aim to bring an understanding of the concept of software reliability and to impart some ideas, which should lead to the development of more reliable software systems. The book tries to bridge the gap between theory and practice and will thus be valuable supplemental reading for a course on software engineering.

During my work on real-time systems in industry I have seen many occasions where the subjects of testing, error detection and error handling have been tackled in an unsystematic, *ad hoc* fashion which leads to subsequent problems in the integration phase. The chapters on these subjects formed the starting point of this book and other chapters were developed in order to produce a clear and concise text covering the whole subject of software reliability, while always keeping the practical aspect in mind.

This English edition is a revised version of the original German edition. The author would like to thank all his friends and colleagues for their help, suggestions and remarks on the German text. Many of the comments which have been raised have been considered in this revised English version.

Particular thanks go to Mr Williams, for the assistance in the translation of the manuscript, and to the editor of Carl Hanser Verlag, Mr Spencker, and the editor of Macmillan, Mr Stewart, for their kind co-operation. Above all, I would like to thank my wife Renate for her constant encouragement and help.

Berlin,
January 1979

H. KOPETZ

Contents

1 *Introduction*

> 'As long as there were no machines, Programming was no problem at all; when we had a few weak computers, Programming became a mild problem, and now we have gigantic computers, Programming has become an equally gigantic problem.'
>
> E. W. Dijkstra (1972b) p. 861

With the first generation of computers the problems in programming were blamed on the severe constraints imposed by the hardware of that time. Since then there have been tremendous technological advances in the field of computer hardware. But although the hardware has become much more flexible, the problems with programming have not decreased; on the contrary, they are worse now than they ever were before. The physical constraints of the hardware have been replaced by the invisible constraints of the capacity of the human mind. The neglect of these psychological limits, together with a disquieting optimism, has led to the design of large software systems. It is not until these very complex logical systems are realised that the incompleteness and inconsistencies of the human intellect show up and result in a number of errors. Each one of these errors can be considered as a single logical flaw without any relation to the whole. Experience shows, however, that all these errors, if seen as a whole, describe a general phenomenon commonly referred to as the 'software crisis'. The complexity of many software systems has become unmanageable, and the natural consequences are delays and unreliability.

There are considerable economic implications connected with the unreliability of software. Between one-third and one-half of the effort that goes into the development and maintenance of a software system is spent on testing and debugging. Since during the implementation of a large computer system, more resources are allocated to the software than to the hardware, the direct costs of the software unreliability themselves amount to a substantial fraction of all computer costs. If, in addition, the indirect costs of errors (for example, lost benefits of a system) are considered, then the economic significance of software reliability becomes even more pronounced.

There are a number of methods by which more reliable software systems can be produced.

(1) By a design methodology which leads to a highly reliable product. If we were successful in finding such a design methodology we could completely eliminate – in theory at least – all activities which are connected with

testing, debugging and run-time error treatment. This alternative, the constructive approach to software reliability, precedes all other methods and must be considered the most effective approach to software reliability. In the past few years, a considerable amount of effort has been spent in the development of improved software design techniques. Although some promising results have been achieved it is to be doubted that a design methodology in itself is sufficient for the development of software systems of the required reliability.

(2) By testing and debugging. This method assumes that the required reliability of a software product can be achieved by very thorough testing and debugging. At the present time more effort is spent in the testing and debugging phase than in any other phase of the software development process. In spite of this, the problem of the unreliability of software has not been solved satisfactorily. This can be attributed to the fact that, even during a very thorough test, only a small fraction of all possible input cases of a software system can be executed.

(3) By the inclusion of redundancy in order to detect and correct errors which show up during the use of a software system. This last method is distinctly different from the previous methods. It is assumed that a complex computer system will always contain errors and steps are taken to reduce the consequences of such errors.

This book is based on the assumption that reliable software can be developed most effectively by a combination of all three methods. The main emphasis is put on software for on-line, real-time systems, since in these systems the consequences of errors and failures are normally much more serious than in systems for batch processing.

2 *Basic Concepts*

2.1 RELIABILITY

Every technical system is developed with the intention of fulfilling a particular function. A measure of how well this function is performed is given by the capability of the system, which does not normally give any indication of the period for which the system runs without cause for complaint. The capability as a function of time, depends on reliability and maintainability.

The systematic investigation of reliability starts with the realisation that the reliability of a system may be defined as a probability.

The reliability of a technical system is the probability that the system performs its assigned function under specified environmental conditions for a given period of time.

It is synonymous with the probability of survival of a system. Quantitatively it may be described by a reliability function, $R(t)$, which gives the probability that a system will function over the time interval $(0, t)$. It has the following properties.

$R(0) = 1$; the system is certain to function at the beginning of the interval.

$R(\infty) = 0$; at time $t = \infty$ the system is certain to have failed.

In the interval $(0, \infty)$ the function decreases monotonically.

The unreliability $Q(t)$ of a system is defined as the probability of failure, hence

$$Q(t) + R(t) = 1$$

Next the concept of effectiveness is introduced. A system is said to be effective when it not only performs its allotted task, but also operates over a long period. The effectiveness as a function of time is not only dependent on the reliability, but also on the maintainability (figure 2.1).

The maintainability of a system is the probability that, after the appearance of an error, the system is returned to an operational condition in a given time. The average time taken to correct an error is known as the 'Mean Time to Repair', abbreviated to MTTR. This interval starts at the appearance of the error and

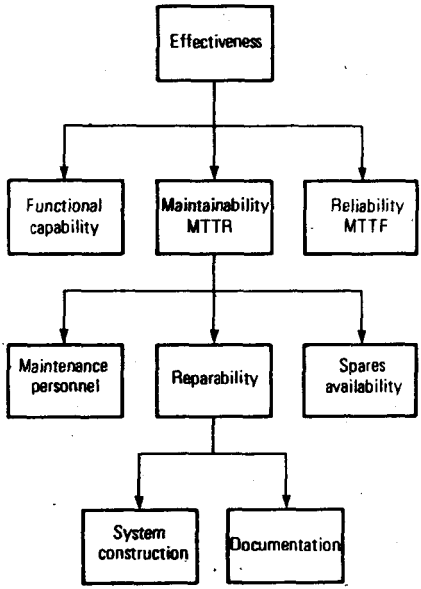

Figure 2.1 Interrelationship of functional capability, maintainability and reliability

ends when the system becomes operational again. Maintainability depends on a number of factors; the availability and competence of maintenance personnel, the availability of spare parts, and the ease with which the system may be repaired, that is, reparability.

The reparability of a system is the probability that an error will be repaired in a given time, by service personnel of average ability, assuming that spare parts are available. While maintainability is also a function of the service organisation, reparability is a system characteristic, that is, it depends on system construction, documentation, and so on.

Of interest in practical applications is the Mean Time Between Failures (MTBF). Strictly speaking, such a mean only has any relevance if the failure rate over a long period (in comparison to the MTBF) does not change. In such a case the conditional probability of failure during any given interval is constant. (The condition is that the system is functional at the beginning of that interval).

The MTBF is made up of two terms, the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR) that is

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Since the MTTR is generally small in comparison to the MTTF, MTBF is often in practice taken to be synonymous with the MTTF.

System availability is defined as the percentage of the time, during a given

interval, that the system is in fact available, thus

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

A high availability can be achieved by a very small MTTR compared with a relatively short MTTF, as well as a longer MTTR for a larger MTTF (figure 2.2). The availability alone is thus not sufficient to characterise completely the relationship between effectiveness and time; in many cases the actual number of failures may be the decisive factor.

Redundancy

If a system contains more resources than are absolutely necessary for the fulfilment of its task, it is said to contain redundancy. The term 'resources' is here taken in its broadest sense, and in a computer system can mean hardware, software or time (Avizienis, 1972).

As a measure of redundancy the following relationship is introduced (Neumann *et al.*, 1973, p. 15).

$$r = \frac{\text{additional resources}}{\text{minimum necessary} + \text{additional resources}}$$

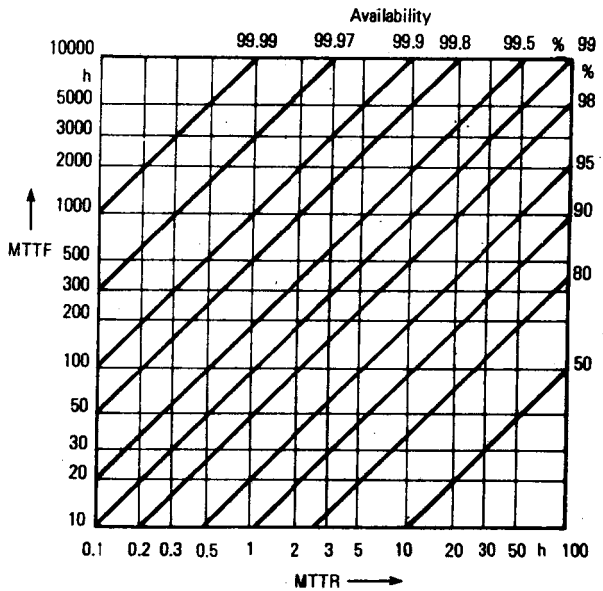

Figure 2.2 Relationship between maintainability (MTTR), reliability (MTTF) and availability

Redundancy is necessary for the detection and correction of faults. During the design stage of a system it is possible to choose between two completely different alternatives in order to increase the reliability of a system. High reliability may be achieved either without redundancy by using high quality components, or by using a greater number of average quality components redundantly.

Active redundancy is employed when the redundant components take an active part in the normal system operation. Standby redundancy is employed when the redundant system parts are only switched on when the active parts of the system drop out due to a failure.

If it has been decided to increase system reliability by means of redundancy, then a choice has to be made between redundancy at component or at system level. For the same amount of additional resources, redundancy at the system level offers a smaller increase in reliability than at the component level. The use of redundancy at component level, however, places greater demands on the engineer. Protection against error propagation in systems with redundancy at component level requires a greater development effort than in systems with redundancy at system level.

2.2 PROGRAMS AND PROCESSES

The terminology used thus far has been developed for, and applies primarily to, the reliability of equipment, that is, hardware. Implicit in this is the assumption that a system that functions correctly initially will eventually fail due to the ageing of its components.

The situation is somewhat different with software. There will be no failures due to ageing, failure will be due to design errors. A design error in part of a program will first lead to observable symptoms when that part of the program is actually executed with appropriate input data. Thus, the relationship between the static program text and the dynamic execution of the program is of great importance when dealing with software reliability. The following model has been substantially influenced by Denning (1971), Dijkstra (1972a), Goos (1973), Horning *et al.* (1973), and King (1971).

A program may be thought of as an ordered set of instructions or statements

$$\langle S_1, S_2, S_3, \ldots, S_m \rangle$$

The instruction space of the program is made up of elements of this set.

The execution of a single statement is called an action or elementary process. An action can be considered under two aspects, the transformational aspect and the control aspect.

The transformational aspect deals with the transformation of information stored in the computer memory. Each action can be considered as a function operating on a number of variables

$$x_1, x_2, \ldots, x_n$$

which assume input values from the domains

$$D_1, D_2, \ldots, D_n$$

and results from the ranges

$$R_1, R_2, \ldots, R_n$$

The Cartesian product

$$D = \underset{i=1,n}{X} D_i$$

is defined as the input space

$$R = \underset{i=1,n}{X} R_i$$

as the output space and the union of the input space and output space as the data space of the action. The variables are called input variables and output variables of the action respectively.

The control aspect of an action deals with the definition of the successor statement. There are two types of binary relations between statements. Statement 2 is the immediate successor of statement 1 and statement 2 takes as input the result of statement 1. The integer variable, which designates the successor statement is called the program counter j where

$$1 \leqslant j \leqslant m$$

In our model we distinguish between three kinds of executable statements.

(1) The execution of an assignment statement (assignment action)

$$x_k, f(x_1, x_2, \ldots, x_n), j$$

causes the value of the variable x_k to be changed according to the result of the function f. The successor statement to be executed is the immediate successor in the instruction space.

(2) The execution of a test and branch statement (test and branch action)

$$P(x_1, x_2, \ldots x_n),\ j_1, j_2$$

involves the examination of a predicate P over the input variable which is either true or false. If it is true, the successor statement is j_1, otherwise j_2.

(3) The execution of a halt statement (halt action) terminates the execution of the program.

At certain well defined instants of time the state of the sytem can be described by a state vector

$$v = \langle a_1, a_2, \ldots, a_n, N \rangle$$

where $a_1, a_2, \ldots, a_n, N$ define the values of the program variables $X_1, X_2, \ldots, X_n$, and the program counter respectively. The set of all possible states of the state vector is the state space of the system.

We call a program a program module if it contains one and only one halt statement, if the last element of the ordered set of statements is this halt statement and if the instruction space is not modified during the execution of the program. It is assumed that the execution of a program module is started at the first statement of the instruction space. In the following the terms program and program module will be used synonymously.

The relationship between a program and its execution can be described on four different levels of abstraction.

(1) Given an initial state vector v_0 we consider the sequence of state vectors

$$v_0, v_1, v_2, \ldots, v_r \quad .$$

generated during the execution of the module. This sequence of state vectors is called a computation generated by this module. The control path of a computation is defined as the sequence of values of the program counter.

(2) If we abstract from the specific data transformation aspect and consider only the sequence of actions

$$a_1, a_2, a_3, \ldots, a_r$$

performed during the execution of the module, provided there is an initial state vector v_0, we get an action sequence of that module. A decision-free program module can generate only one action sequence but many different computations.

(3) If we abstract from the specific data transformations and from the specific action sequences generated during the execution of the module and consider only the general data transformations under the assumption that the execution will start at the first statement and will terminate, we get the process which is generated by the execution of the program module. Each process is embedded in an environment which furnishes the inputs and uses the results. Any variable that is an input variable to any one action of the process is a significant variable of the process. Any variable that is an out-

put variable to any one action of the process is a changed variable of the process. An input variable of the process is every variable that is a significant variable of the process and a changed variable of the environment. An output variable of the process is every variable that is a changed variable of the process and a significant variable of the environment. All variables that are used in any one action of the process and that are neither input nor output variables of the process are internal variables of the process.

(4) If we abstract from the internal variables of the process and consider only the input variables, the output variables and the data transformation of the process, then the process can be considered as an elementary process or action on a higher level.

We thus have a recursive feature in our definitions. The execution of a program module can be considered as a computation, a sequence of actions, a process or an action on a higher level. The primitive element in our system is a basic data transformation. The above definitions are not necessarily restricted to the software levels. They can also be used to describe hardware processes. There are many common elements between hardware and software processes and in some respects a distinction is not justified.

Due to the recursive nature of the above definitions, the error analysis of a software system with a hierarchical structure may be reduced to the investigation of the behaviour of a single process.

2.3 CORRECTNESS, RELIABILITY AND ROBUSTNESS

It is assumed that no changes will be made to the software system, the correctness and reliability of which is being analysed. Clearly, every change to a software system creates a new system which will have different reliability properties from the original system.

Software Correctness

Software correctness is concerned with the consistency of a program and its specification. A program is correct, if it meets its specification; otherwise it is incorrect. When considering correctness it is not asked whether the specification corresponds to the intentions of the user.

Software Reliability

Software reliability, on the other hand, may only be determined when the actual utilisation of the software by the user is taken into consideration

> 'Reliability is (at least) a binary relation in the product space of software and users, possibly a ternary one in the space of software and users and time.' (Turski, 1974, p. 15.)

In the preceding section the relationship between a program and the process associated with it was discussed. A process may also be thought of as the realisation of a function that relates every point in the input space to a point in its associated output space. If there is an error in the process this realisation will not be achieved as intended, that is, various points in the input space will not produce the expected results. If a procedure is available (a so-called 'test oracle'), whereby it may be determined for every point of the input space whether the computation starting at this point delivers a result that corresponds with the intentions of the user or not, then we can introduce a binary error function $e(i)$ which is defined over the entire input space so that

$e(i) = 0$ the computation, that starts at point i, is correct
$e(i) = 1$ the computation that starts at point i is either incorrect, or does not terminate where $i = 1, \ldots, N$
and N = the number of points in the input space (MacWilliams, 1973)

For every application there is defined, over the complete input range, the distribution of input values such that

$\Sigma p(i) = 1$
$p(i)$ the probability that point i occurs in the input domain for the particular application

The probability of the appearance of a software error is then given by

$\lambda_n = \Sigma e(i)p(i)$
λ_n = probability of the appearance of a software error for an input case (Kopetz, 1974).

The choice of a particular input case, from the set of all possible input cases, is the desired random event. Software reliability may thus be defined as follows.

> The reliability of software is the probability that a software system fulfils its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and input are free of error.

In software systems, the selection of the input completely determines the output of a computation. Herein lies the fundamental difference from hardware reliability. With an error caused by ageing, it is not necessary to look at a particular input, since it can be assumed that the hardware originally functioned for all inputs, and the error lies in the breakdown of a hardware component (Avizienis, 1972). (The problem of design errors in the hardware logically belongs to the class of software errors.) The failure rate of software can only change when either the error function changes, that is, changes to the software are carried

out, or the input distribution changes, that is, the program is placed in a new environment. An example of the varying error rates due to a changing input distribution is encountered in the commissioning of a software system. As soon as the test conditions change, the error rate changes by leaps and bounds (Wolverton and Schick, 1972).

If it is assumed that the input distribution does not change, and that the program is not modified, then the software error rate cannot change. Under these conditions software errors may be described by a constant error rate.

The relationship between the reliability defined as a function of time, and the probability of error for each of the input cases may be written in terms of the system input rate

$$\lambda_t = \lambda_n r$$

where λ_n = probability of the appearance of a software error for an isolated input case

λ_t = error rate as a function of time

r = number of input cases per unit time (input rate).

The same reasoning can also be applied to real-time systems. The time axis can be subdivided into discrete intervals that are defined by the cycle time of the computer. Timing starts from a defined initial condition of the system. Clearly the input range must be expanded with the time dimension. This additional dimension increases the complexity of a system significantly, as is known from common experience.

In a number of publications (Jelinski and Moranda, 1972; Shooman, 1976; Hamilton and Musa, 1978) reliability growth models for software have been presented. These models not only consider the reliability of a software system *per se* but look also at the change (hopefully growth) of reliability as a consequence of program modifications (error elimination). In the development of these models many assumptions about failure mechanisms, number of residual errors, program modifications and the use of the system have to be made. It has yet to be shown whether all of these assumptions are fully justified (for a criticism of some of these assumptions see Littlewood, 1978).

Software Robustness

The concept of software robustness is used to investigate the relationship between software and system reliability.

System reliability may be defined as the probability that the computer system performs its allotted task during a given period of time, under specified environmental conditions.

The term environmental conditions is taken to mean not only the physical environment surrounding the computer, but also the input distribution (and its rate) and the average number of input errors.