Defence Science Journal, Vol 47, No 1, January 1997, pp. 19-26
© 1997, DESIDOC

Reliability Analysis in Design & Development of Checkout Software for Aerospace System

J. Chattopadhyay
Research Centre Imarat, Hyderabad - 500 269.

ABSTRACT

Checkout system is the nerve centre for test, maintenance and launch of an aerospace vehicle. Real-time software embedded in it plays a vital role in taking decisions at critical point of time. The problem of not detecting a fault in the system under test or raising a false alarm due to the misinterpretation of a parameter will lead to crashing of time, bouncing on safety and crunching on cost. It is therefore essential to develop and deliver a fault-free software for this system. Software reliability is defined as the probability that it will work without fail for a specified period. Unlike hardware, it is difficult to count and predict the failures and thereby generate a figure of reliability. In case of checkout software, activities are more dependent on the environment and the parameters are also random with respect to time. This paper highlights the set of procedures evolved during the process of designing and delivering various reliable checkout software for aerospace system. This has given an insight into the failure analysis and has shown the correctness of software under all foreseeable conditions. Basic execution time model is used to measure the desired software reliability.

1. INTRODUCTION

Checkout system (CS) is an integral part of aerospace system. Flight worthiness is declared and assessed using this system. Aerospace system reliability should be as high as possible at the cost of space and weight. This dictates onto the techniques of fault-tolerance adopted to improve system reliability and also the reliability of CS. With the advancement in technology, aerospace system today has a large number of sophisticated units which are to be tested with utmost care and high degree of technical support. Thus, testing and clearance of aerospace system has become critical and complex. Specially-designed CSs with a large number of hardware and software components have come up to fit into this requirement.

System reliability is desired both in hardware and software. A false alarm or an unnoticed problem can cause a massive failure in different perspectives. The sources of failure in software are the design faults, while the principal source in hardware is physical deterioration[1]. Once a software (design) defect is properly fixed, it is in general fixed for all time, except in certain cases where the defect is due to some unforeseen environment which was not tested or developed for[2]. The probability of hardware failure due to wear and other physical causes has usually been much higher than the one due to an unrecognised design problem. Thus, hardware reliability has a specific pattern and can be altered with the use of definite redundancy. On the other hand, software reliability is due to the inherent faults that are not visible and are very difficult to assess. Moreover,

Received 28 October 1996

it also changes or improves the process of development and maintenance. Thus hardware and software reliability follow inverse logic.

This paper concentrates on the software reliability aspects of CS designed for an aerospace system. Software reliability is due to design faults or environmental problems and is related to software life cycle. The figure of reliability is improved at every phase. To design a reliable software, care should be taken at each point of the life cycle. The best way of addressing this problem is to model software reliability with the primary factors, like fault introduction, its removal and the environment2. Data is collected on different systems with a major thrust on a particular CS already deployed. The parameters for a basic execution model are evaluated and applied to predict the-system reliability for given CPU hours.

2. CHECKOUT SOFTWARE OVERVIEW

The CS is based on a real-time operating system to perform data acquisition, control and decision making operations. It has the following main modules:

(a) Unit test module (UTM)

(b) Auto sequencer module (ASM)

(c) Checkout libraries module (CLM)

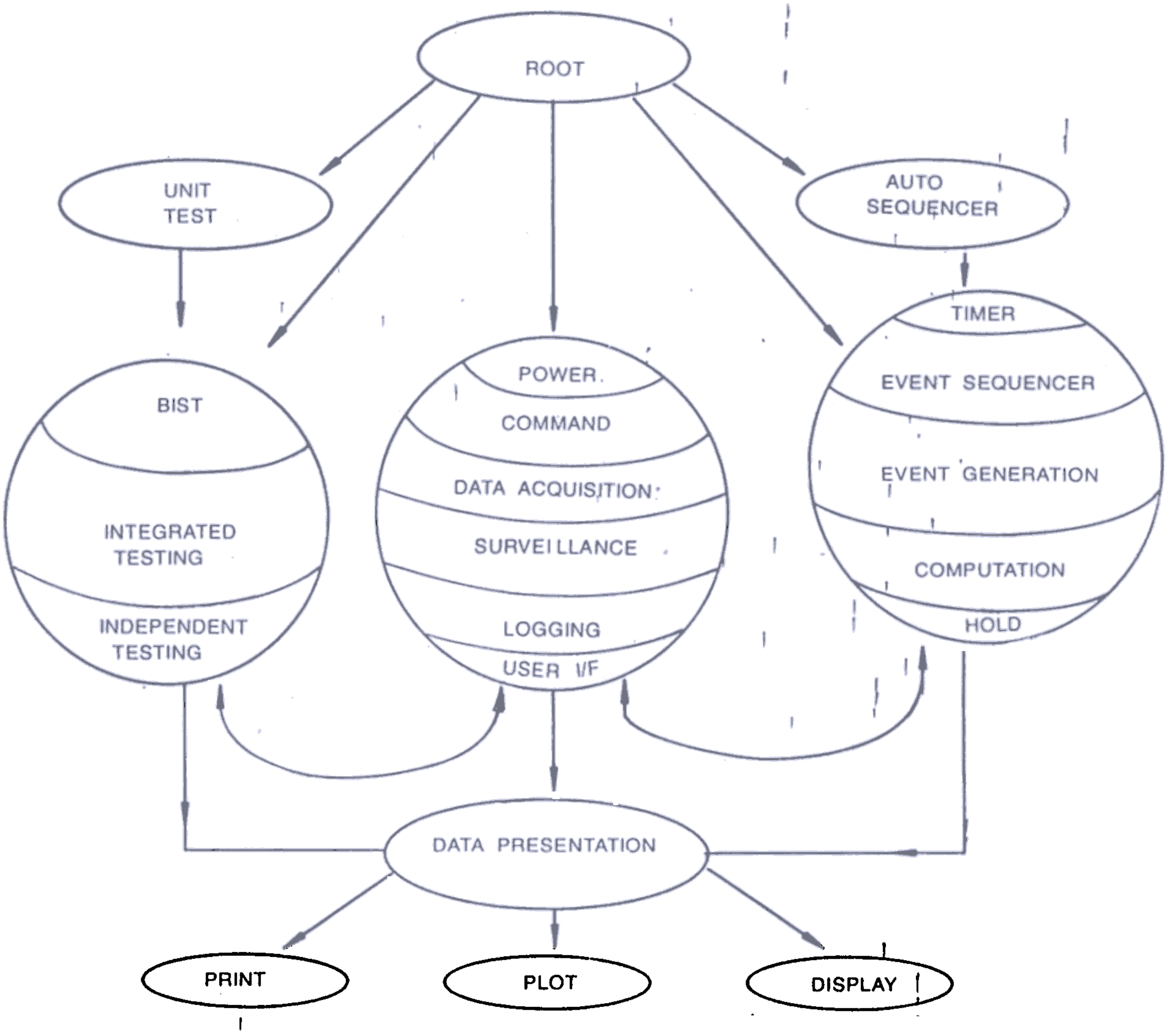

Figure 1. Checkout software

Each one of these has different sub-modules and sub-sub-modules as shown in Fig. 1.

The UTM may have multiple modules depending on the number of sub-systems that can be electrically checked in an aerospace system. Each sub-system has built-in self test (BIST) to indicate its health. The sub-systems are to be tested both in integrated and independent modes to repair the target system by replacement. Each UTM is capable of exciting the relevant sub-system and perform its functional test at different configurations.

The ASM is a time-based routine with event interlock. It performs last minute checks before take-off and also initialises some of the systems to ensure the lift-off, and the settings that ensure correct flight. Besides, it does surveillance on each parameter through multiple monitoring points. This is the most critical part of checkout software. During this process, if an eventuality is caused due to a parameter, it can quickly reset the system and generate 'hold'. The last minute criticality determines the fate of the flight. Hence, the software at this phase has to be highly reliable.

There are software modules known to be CLM, related to different input-output link, data acquisition, engineering conversion, computer communication and command actuation. Other than this, there are modules for user interface for menu, graphics display, plot and print. Besides, there are modules that are embedded within the flight systems which are to be used for ground checkout. This is specially applicable to intelligent units. The CS is required to initiate these routines and interpret the results. Depending upon the type of the system and the numbers of sub-systems, there will be growth in the software modules. However, many modules are common. Failure data is generated on several CSs. The system that is being presently considered has 200 modules and about 400 K code and data size.

3. SOFTWARE DEVELOPMENT PROCESS

Over the years, several software development processes have been evolved to bring down the design errors and also to make the whole activity visible so that the change control and quality control can be implemented. Royce model was published in late sixties. The Waterfall model and 'V' model followed later on. However, these are strictly for software development process. The CS has a dedicated computer environment, and the software has to be designed and developed along with the hardware. In this case, it is essential that the approach to reliability takes the total system in picture. Department of Defence Standard DOD-STD-2167 and WINGROVE[2,6] development cycle fits in this requirement. Reliability is ensured by maintaining a correct flow which is reviewed at each stage using the following techniques:

* Management - configuration control
* Specification optimisation
* Structured programming technique
* Segmented or modular program development
* Design reviews and walkthroughs
* Use of flow charts, heirarchial input process output (HIPO), Pseudo code prior to code
* Top-down development
* Visibility of software by documentation and code comment
* Testing by designer
* Validation on testing
* Testing in a working environment prior to use
* Use of fail-safe, fault-tolerant software
* Use of effective validation and test procedures, stage-wise
* Use of numerical techniques and models to quality software reliability.

4. FAULT-TOLERANCE IN SOFTWARE DESIGN

The techniques that are to be followed to design the software have already been discussed. Collectively, these techniques attempt to prevent the existance of faults in the operational software. But in case of realistic systems, they are unlikely to be totally successful and a number of residual faults still remain. It will be appropriate to supplement fault-prevention with design approaches which attempt to suppress the effects of residual faults. However, this scheme preserves the structural quality and the coupling requirements[1].

A single abstract model to describe a software system consists of a number of components that cooperate under the influence of a design to meet the demands of the system environment (Fig. 2). The design can be considered as the algorithm which is responsible for defining interactions

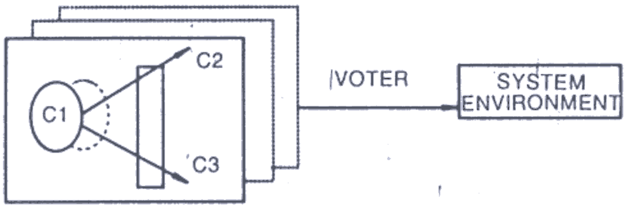

Figure 2. Software fault tolerant model

between components and system environment. The objective of the software fault-tolerance is to prevent the software faults from causing system failure. A component redundancy with a voting mechanism to determine the system response may be an answer to this problem. However, this has the necessary overhead and structural complexity. The CS in which the response time is a critical issue has to select a technique that keeps alive its performance level. To achieve this, the following two methods are considered.

(a) N-version programming

(b) Recovery blocks.

4.1 N-Version Programming

This is achieved by utilising three or more versions of a program each of which has been independently designed using the same specifications and is activated by a driver module that controls all input-output data and determines overall output through a majority voting scheme[4]. The present version of CS is a two-version programming. The voting decision is taken by the operator so that there is no overhead (Fig. 3).

Figure 3 shows the existence of common modules between ASM and UTM. These are related to the surveillance of critical parameters which calls for a HOLD in case of an abnormality. These modules are duplicated and can be selected on the operator's choice. A provision is made to take the final decision based on the reporting of both the

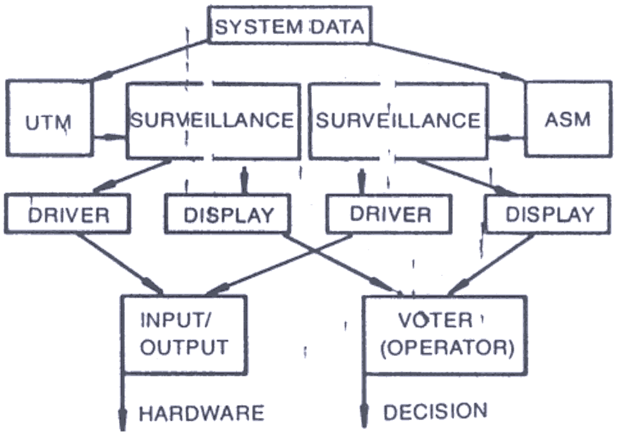

Figure 3. Two-version checkout program

modules. The efficiency of this scheme has been appreciated in the fault finding on different occasions without overhead components.

4.2 Recovery Blocks

In this method, a number of blocks using the same specifications are designed. If primary block does not function, the job is allotted to the other block and is thus continued; and the decision is passed on to an acceptance module[1]. However, due to the large overheads, a scheme like functional degraded alternates with rollback recovery facility is chosen. Primary module provides full functionality, whereas alternate modules provide progressively degraded functionality, being an old version. In case of checkout software, due to inherent fault problem, there may be a spurious hold generation at particular point. A provision is extended in this case through (i) multipoint monitoring, and (ii) by-pass scheme.

The multipoint monitoring refers to the monitoring of the same parameter through telemetry link and also through direct link to the sensors so that in case of a problem, the acceptance module can use either link.

The by-pass scheme is employed through the provision of manual hold. In case of detection of incorrect reading of a parameter, it forces the system to previous best point so that the process can continue with the faulty parameters. Decisions for hold generation is taken by the operator; therefore, the system works with degradation. For example, during a control system check, the allied parameters are: hydraulic pressure, battery voltage, battery current and feedback voltages. There may be a case, where it is found that all other parameters are working normally except the current monitoring. It generates hold. It is inferred that the system health is normal, and there may be problem with the current monitoring module. Hence, the launch can continue. This facility is called the rollback recovery, which is the capability of the system to return to the consistent state[7], that existed before it failed. In this case, the system calls for a hold with a provision to lift the hold and proceed from the same point and the system is restarted.

5. TEST & VALIDATION SCHEMES TO BUILD RELIABLE CS

Validation, verification and testing are the three terms which finally contribute to the removal of design faults and thereby generate a reliable software. A term often comes out as ready-to-release to determine when to stop testing and also the reliability criteria it should meet. Several definitions are available for testing by Hetzel and Myers. However, the following definition is the best fitted in the present context.

'Testing is the process that satisfies the reliability requirement to be achieved on its deployment in operation phase'. In general, the steps followed in testing are

(a) Testing through reviews.

(b) Unit testing

(c) System testing

(d) Acceptance testing.

Reviews are part of software life cycle. They are conducted at the completion of each phase of the development cycle. Unit testing is achieved through an external (black box) perspective with test cases based on the specifications of what the program is supposed to do, or on an internal (white box) perspective with test cases developed to cover or exercise the internal logic. During system testing, all the modules are brought together. Mostly, it is the black box. This phase is completed on the basis of number of errors/faults encountered. In general, this activity is conducted to perform functional testing, structural testing and also testing for the correctness of proof.

In case of CS software design, this is the most crucial phase where the maximum number of design faults are eliminated. A specific method is adopted to achieve a given fault intensity target. Software validation steps are shown in Fig. 4.

Failure data is collected from Step 3 for reliability measurement. Necessary data as regards to failure intensity and the total number of failures are estimated at this stage. The target reliability is achieved through Step 4. The design of the simulator to test the system largely depends on this activity. A good simulator design is also a part of checkout system design. Normally, 10 hours per day for 30 days of testing is recommended at this

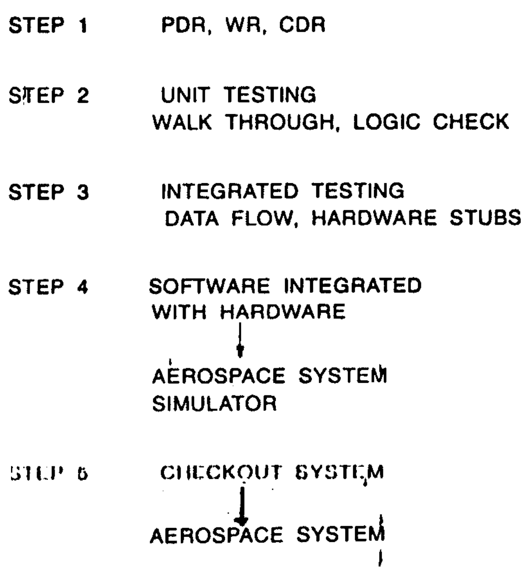
STEP 1
PDR, WR, CDR
STEP 2
UNIT TESTING
WALK THROUGH, LOGIC CHECK
STEP 3
INTEGRATED TESTING
DATA FLOW, HARDWARE STUBS
STEP 4
SOFTWARE INTEGRATED WITH HARDWARE
AEROSPACE SYSTEM SIMULATOR
STEP 5
CHECKOUT SYSTEM
AEROSPACE SYSTEM

Figure 4. Software validation steps

phase. Finally at Step 5, the checkout system is connected to the actual hardware and evaluated for its performance. There may be a cyclic operation between Step 4 and Step 5. These schemes are evolved with a lot of iterations and on application to different aerospace systems.

6. CHECKOUT SOFTWARE RELIABILITY MODELLING & PREDICTION

Establishing reliability is a major challenge in software production environment. A software product can be released only after some threshold reliability criterion has been satisfied. The most useful parameters are : residual fault density and failure intensity. Software reliability models are a recent concept. They were brought into effect by Jelnski and Moranda, Littlewood[5] and Verall, Shooman & Musa, during seventies and by Musa and Okumoto, Dale in eighties. Since then, various models have come up and are being used in different areas.

For software reliability model, one must consider principal factors that affect it, viz., fault introduction, fault removal and the environment. Fault introduction depends primarily on the developed code and development process characteristics. The most significant code characteristic is its size. Code can be developed to add features or remove faults. Fault removal depends on the operational profile. Since some of the foregoing factors are probabilistic in nature and operate over time, software reliability models are generally formulated in terms of random process. The models are distinguished from each other in general terms by probability distribution of failure times of the number of failures experienced during a fixed time interal or by the nature of the variation of rahdom process with time. The possibilities of different mathematical forms to describe the failure process are almost limitless. To choose a particular model, the following points are to be considered: (i) The model should give good predictions of future failure behaviour, (ii) compute useful quantities, (iii) be simple, (iv) widely applicable, and (v) based on sound assumptions.

From the literature, it is found that the basic execution time model is generally superior in capability and applicability to other published models[2]. It is good for the pretest study, i.e. till it attains ready-to-release state. For checkout software, basic execution model is an ideal selection from the parameter estimation point.

6.1 Basic Execution Time Model

The failure intensity λ for the basic model as a function of failures experienced is

$$\lambda(\mu) = \lambda_o[1 - \mu/v_o] \quad (1)$$

λ_o = Initial failure intensity

μ = Failure experienced

v_o = Total number of failures.

The slope of failure intensity

$$\delta\lambda/\delta\mu = -\lambda_o/v_o \quad (2)$$

This confirms that the failure intensity comes down against time. Failure intensity at time (t) is given by

$$\lambda(t) = \lambda_o \exp[-(\lambda_o/v_o)t] \quad (3)$$

and reliability at t hours for a period of t hours is given by

$$R(t'/t) = \exp\{-\left[v_o \exp[-(\lambda_o/v_o)t]\right] \left[-\exp[-(\lambda_o/v_o)t']\right]\} \quad (4)$$

A figure of reliability and failure intensity of a checkout software designed and developed is determined from the above equations. Parameters to be estimated are initial failure intensity λ_o and total failure v_o.

6.2 Computation

Total code size = 300 K.

Average inherent faults found on similar type of software = 3 faults/1K code.

Inherent fault = 900.

Considering fault reduction factor (B) as unity, total failure, ν_b = 900.

A data table is generated during software integration time based on the number of failures *vs* CPU hours. It is represented in Table 1.

Table 1. Time vs failure data

Time (hr)	1	2	3	4	5	6	7	8	9	10
Failures	30	42	26	38	22	28	30	19	22	17

Initial failure intensity λ_o can be estimated as 27.4 from Table 1. As discussed in Step 4 of system validation, a run of 300 hr is given to obtain the desired software reliability. Checkout software is required to function continuously for 5 hr. Failure intensity is 0.0029559 failures/CPU hour. Software reliability at the time of actual testing is 0.986368.

6.3 Observations

1 Reliability figure and failure intensity can be improved on further testing. Since it was in the accepted band, the software was released for operation.

2. It shows that the failure intensity decays exponentially with time, whereas software reliability improves. This is shown in Fig. 5 by generating data at different testing times.

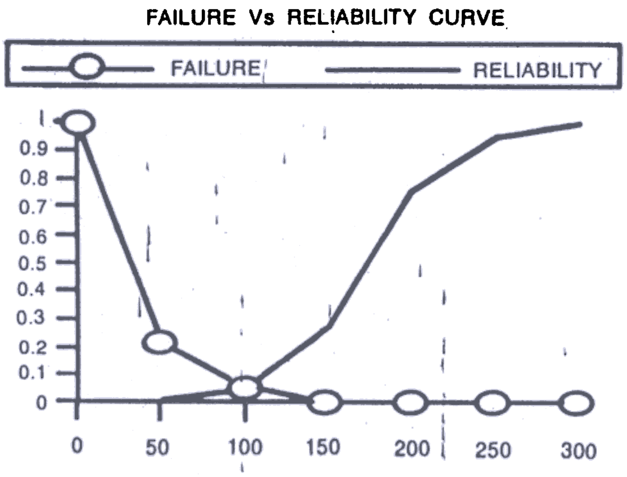

Figure 5. CPU time/hour

3. It is inferred that good simulation aid and testing methods increase the initial failure intensity rate and that allows to gain desired system reliability within the shortest time. Again, on increasing the testing period, one can achieve higher order of reliability.

7. CONCLUSION

Most of the techniques related to software development are taken care of. It is shown that a target reliability figure always helps to streamline the software design, techniques, testing methodologies and time to convert it to 'ready-to-release' state. An attempt has been made to establish theories in practice. Some assumptions are made relating to the test data and their computation. A better estimation technique has to be adopted. Basic execution time model has a constant slope, which contradicts with the practical data. Reliability increases with the faults repaired, but the effect of repair on fault growth is not considered. So there may be a difference of opinion in reliability prediction. It is recommended to use the Poisson's logarithmic process. However, a good approach to establish reliability in software and their measurement at primary level is brought out clearly and distinctly. This can be used as a yardstick for future work.

REFERENCES

1. Paul, Rook. Software reliability handbook. Elsevier Applied Science, London.

2. John, D. Musa.; Anthony, Iannino & Kazuhira, Okumoto. Software reliability, measurement, prediction, application. McGraw-Hill Book Company, New York.

3. Bil, Hetzel. The complete guide to software testing. QED Information Science Inc, Wellesley, Massachusetts-02181.

4. Anderson, T. & Knight, J.C. Frame work for software fault tolerance in real-time systems. *IEEE Trans. on Software Engineering*, 1983, SE-9(3), 335-64.

5. Littlewood, B. Software reliability. Blackwell Scientific Publications, Oxford, London.

6. Robert, N. Charette. Software engineering environments concepts and technology. Intertext Publications Inc., McGraw-Hill, Inc. New York.

7. Yashwant, Malaiya, K.; Karunanithi, Nachimuthu & Verma, Pradeep. Predictability of software reliability models. *IEEE Trans. on Reliability*, 1992, **41**(4), 539-46.

8. Norman, F. Schneidewind. Software reliability model with optimal selection of failure data. *IEEE Trans. on Software Engineering*, 1993, **19**(11), 1095-104.

9. Kitchenham & Littlewood, B. Measurement for software control and assurance. Elsevier Applied Science, London.

Contributor

Mr J Chattopadhyay received AMIE in Electronics from Institution of Engineers and MTech in Automation and Control from JNTU, Hyderabad. He joined DRDO at Research Centre Imarat in 1986 and has been working in the design and development of checkout and launch systems for IGMDP. At present, he is working as system manager and chief designer of checkout systems for different missile projects. He has gained expertise in the design of real-time systems for embedded applications. Before joining DRDL, he completed one-year Electronics Fellowship Course at IAT, Pune. He is member of the Institution of Engineers.