International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 5, July - 2012

Reliability Allocation for Fault Tolerant Software

Krishna Kumar Singh[1] and Dr. S. K Chaturvedi[2]

[1] Lecturer in CSE Department, RGU IIIT, Nuzvid
[2]Associate Professor in Reliability Engineering Centre, IIT Kharagpur

Abstract

Fault tolerance is one of the major concerns in software design nowadays. In this paper a simple optimization model for the fault tolerant software reliability allocation using fault tree and event tree techniques incorporated with cost (testing time) minimization approach is presented, which takes software module complexity (size) into account along with its criticality with respect to other modules.

A simple methodology for the fault tolerant software reliability allocation model is presented mainly based on two methods: cut set method (SFTA) [7], and cost (testing time) minimization, where objective function is derived from Musa basic execution model and Musa-Okumoto logarithmic model.

1. Introduction

Reliability Allocation deals with the setting of reliability goals to individual Software module or component, so that a specified reliability goal is met. The apportionment of reliability values among the various components can be made on the basis of complexity, occurrence, criticality and utility.

Several papers have addressed the problem of software reliability allocation. Zahedi and Ashrafi [4] modeled Software Reliability Allocation Based on Structure, Utility, Price, and Cost within a nonlinear programming formulation that maximizes system reliability subject to a cost constraint; Misra [10], proposed a cost model for allocation of component failure intensities to achieve a software systemꞌs reliability target while minimizing the cost in the design phase.

The above two papers highly rely on expertise and experience to estimate the parameters used in the model; R. Lyu [6], has taken the initial failure rate of each software module as one of parameter to minimize total cost (testing time) and considered all modules are connected in logically series fashion. Malaiya [5] presented a allocation model to minimize cost subject to an overall system failure intensity goal, same as Lyu [6] but considered that execution frequency also affects the allocated reliability; Xiang [7] presented a fault tree analysis of reliability allocation. He considered the structural complexity (redundancy), but not internal complexity (e.g. size) of software modules; Lyu and Sampath [15] presented the idea regarding fault tolerant software reliability allocation based on coverage factor, but it can be estimated during integration testing phase. The optimization of reliability allocation, subject to reliability constraints (derived from fault tree or event tree), considers the internal complexity (e.g. size of module) as well as structural redundancy and criticality.

Three of the best-known fault-tolerant software design methods are N-version programming (NVP), recovery block scheme (RBS), and N-self-checking programming. All three methods are based on the redundancy of software modules (functionally equivalent but independently developed) and the assumption that coincident failures of modules are rare. This approach presumes the execution of N functionally equivalent software modules (called versions) that receive the same input and send their outputs to a voter, which is aimed at determining the system output. The voter produces an output if at least M-out-of-N outputs agrees, otherwise, the system fails.

The rest of the Section is organized as follows. Section 2 elaborates the description of some important terms, which are prerequisites to understand software reliability allocation. Section 3 covers the introduction of three Basic techniques of fault tolerant software, these techniques are: N-version programming architecture, recovery block architecture, and N-self-checking programming.

Section 4 elaborates the problem statements for reliability allocation model. Section 5 concludes the paper work.

2. Definition of some important terms

This Section defines some important terms and terminology, which are required to understand problem model and its formulation for reliability allocation procedures.

2.1 Source lines of code (SLOC) and instruction execution rate(r).

This is number of source instructions (*Is*) [1] excluding commented lines of code. One source instructions may be equivalent to many machine level instructions. Instruction execution rate is processor execution speed i.e. how many instructions are executed in a unit time (CPU time). For example r = 25 MIPS, i.e. 25000000 instructions are executed in one second.

2.2 Failure rate and failure intensity function ($\lambda(t)$).

A failure occurs when the user perceives that a software program ceases to deliver the expected service. The failure intensity function represents [1] the rate of change of the cumulative failure function. The hazard rate is defined as the probability that a failure per unit time occurs in the interval [t , t + dt], given that a failure has not occurred before t *and* instantaneous hazard rate is refers to failure rate function (i.e. probability of failure in the point of time). In this paper, the failure rate and failure intensity function are used interchangeably.

2.3 Inherent fault density (ρ) .

This is number of faults per KSLOC. The estimate for inherent fault density can be based on KSLOC and Function Points. A fault is uncovered when either a failure of the program occurs, or an internal error (e.g., an incorrect state) is detected within the program. The cause of the failure or the internal error is said to be a fault.

2.4 Fault reduction efficiency factor(B).

This is a measure of the proportion of faults removed from code to faults removed plus new faults introduced, and in other word, it is average no of faults corrected per failure. Suggested defect removal efficiencies of software developed on different Levels of the capability maturity model ([1]CMM) [1] are given in the Table 2.1.

Table 2.1. Fault removal efficiency factor corresponding to CMM level

SEI CMM Levels	Fault Removal Efficiency Factor
SEI CMM 1	0.85
SEI CMM 2	0.89
SEI CMM 3	0.91
SEI CMM 4	0.93
SEI CMM 5	0.95

2.5 Fault exposure ratio (K):

It is expected fraction of existing faults exposed during the execution of software application. i.e. number of faults exposed divided by total number of existing faults. In other word, it can be interpreted as the average number of failures occurring per fault in the code during one linear execution of the program.

1 **Note: CMM** is a benchmark for comparative assessment of software development processes. This service mark owned by Software Engineering Institute (SEI), Carnegie Mellon University (CMU), US.

Musaôs default value of K [1] is given by 4.2×10^{-7} , however, it is suggested that the organization determine an estimate of fault exposure based on historical data. Fault exposure ratio, which can be obtained by normalizing the per-fault hazard rate with respect to the software size and the instruction execution rate. Li and Malaiya [2] have suggested that K varies with the initial fault density and have given the estimates $K = \frac{1.2\times10^{-7}}{\rho} e^{0.05\rho}$

where ρ is defect density per KSLOC.

2.6 Average code expansion ratio (Q_x).

It is number of object instructions per SLOC. It is defined as the ratio of executable line of code generated after compilation to that of legal program source code syntax. To compute the number of object instructions I, the number of executable lines of code is multiplied by the code expansion ratio, supplied in Table 2.2 [10]. If real project data is not available then we can use this table, as this provides average value of estimates.

The rationale behind this data is that the relationship between a line of code and a machine instruction varies depending on the language. Also, the relationship between a line of code and a function point also vary with language.

Table 2.2. Code expansion ratio

Programming Language	Expansion Ratio	Mean Source Statements/Function Point
Assembler	1	320
C	2.5	128
Ada	4.5	71
3rd generation lang.	4	80

2.7 Function points.

Function Points are measures of software size, functionality, and complexity used as a basis for software cost estimation [9], and given by an expression.

Function Points = Unadjusted Function Points × (0.65 + 0.01 × Value Adjustment Factor). Determining the unadjusted function point count consists of counting the number of external inputs, external outputs, external inquiries, internal logical files, and external interface files. Determining the value adjustment factor consists of rating system, input and output, and application complexity. Determining Function Points consists of factoring unadjusted function points and value adjustment factor together

After estimating the function points, we can estimate the inherent faults using CMM level [1] (standard adopted for software development process), as given in the Table 2.3.

Table 2.3: Estimation of total inherent defects using function points

SEI CMM Level	Average function point
SEI CMM 1	5 potential, .75 delivered
SEI CMM 2	4 potential, .44 delivered
SEI CMM 3	3 potential, .27 delivered
SEI CMM 4	2 potential, .14 delivered
SEI CMM 5	1 potential, .05 delivered

2.8 Non-homogeneous Poisson process:

A **non-homogeneous Poisson process [16]** is a Poisson process with rate parameter ∂(*t*) such that the rate parameter of the process is a function of time.
The counting process {N (t), t Ó0} is said to be a non-homogeneous Poisson process with intensity function ∂(t), for t Ó 0 if:

(I) N(0) = 0;
(II) It has independent increments; and
(III) It has unit jumps, that is,

P (N (t + h) ï N (t) = 1) = ∂(t) h + o (h) and
P (N (t + h) ï N (t) Ó 2) = o (h). Where, o (h) is very small quantity
In the non-homogeneous case, the rate parameter ∂ (t) now depends on t.
When ∂ (t) = ∂; constant, then it reduces to the homogeneous case.

2.9 Musa basic execution time model.

The Musa basic execution time model [1][14] assumes that all faults are equally likely to occur, are independent of each other and are actually observed. The execution times between failures are modeled as piecewise exponentially distributed. The intensity function is proportional to the number of faults remaining in the program. Failure intensity is function of average number of failures $\mu(\tau)$ experienced at any given point in time (= failure probability) and is given by

$$\lambda(\tau) = fK(N - \mu(\tau))$$

Where,

$$\mu(\tau) = v_0\left[1 - e^{-\frac{\lambda_0}{v_0}\tau}\right]$$

$$\lambda(\tau) = \lambda_0 e^{\left(-\frac{\lambda_0}{v_0}\tau\right)}$$

Where, *f:* is linear execution frequency, and *K* is fault exposure ratio

μ (τ): is average total number of failures during execution time (τ).
$\lambda(\tau)$: failure intensity function.
λ_0: initial failure intensity at start of execution.
v_0: total number of failures over infinite time.

2.10 Musa-Okumoto logarithmic model.

The Musa-Okumoto model [14] is called logarithmic Poisson execution time model, it assumes that all faults are equally likely to occur and are independent of each other. The expected number of faults is a logarithmic function of time in this model, and the failure intensity decreases exponentially with the expected failures experienced. Finally, the software will experience an inýnite number of failures in inýnite time.
Average total number of counted experienced failures (μ) is a function of the elapsed execution time (τ).

The failure intensity is given by

$$\lambda(\tau) = \frac{\beta_0\beta_1}{\beta_1\tau + 1}$$

$$\mu(\tau) = \beta_0 \ln(\beta_1\tau + 1)$$

$$\beta_0\beta_1 = \lambda_0$$

$$\beta_0 = I_s D_{\min}$$

Where, λ_0 is initial failure intensity

$1/\beta_0$ is failure intensity decay parameter.

D_{min} (fault density) takes a value between 2 and 4 defects per KLOC. For an initial fault density D larger than 10 faults per KLOC, they [14] suggest to set Dmin = D_0/3. Like Musaƒs basic execution time model the ñlogarithmic Poisson execution time modelò by Musa and Okumoto is based on failure data measured in execution time. Its assumptions are as follows:
1. At time t = 0, no failures have been observed.
2. The number of failures observed by time *t, M(t),* follows a Poisson process.

2.11 Fault tree analysis (FTA).

Fault tree analysis is a failure analysis [7] in which an undesired state of a system is analyzed using logic relationship among the various components (or events). Fault Tree Analysis (FTA) attempts to model and analyze failure processes of systems. FTA is basically composed of logic diagrams that display the state of the system and is constructed using graphical design techniques. The fault tree is usually using conventional logic gate symbols.

A cut set is a set of basic events whose occurrence causes the system to fail. To get the minimum cut sets from the whole cut sets; use the relationships of the events below to absorb the redundant cut sets.
A+A=A A+AB=A AA=A. Some example we shall see in Section 3.

2.12 Event tree diagram.

Event tree diagram is based on binary logic, in which an event either has or has not happened or a component has or has not failed. It is valuable in analyzing the consequences arising from a failure or undesired event. Event tree analysis is highly effective in determining how various initiating events can result in accidents of interest. Event trees are useful for system-reliability analysis and risk quantification since they illustrate the logic of combination of probabilities and consequences of event sequences.

3. Basic techniques and terms of fault tolerant software

Software failures are caused by errors made in various phases of program development. When the software reliability is of critical importance, special programming techniques are used in order to achieve its fault tolerance. Three of the best-known fault-tolerant software design methods [13] are *N*-version programming (NVP), recovery block scheme (RBS), and N-self-checking programming (NSCP). Here it is considered that failures of versions of each component are statistically independent and having no coincident failure and hardware faults has not been taken into account. The term coincident failure refers that the two or more functionally equivalent modules fail on the same input case.

3.1. N-version programming (NVP) architecture

In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but hopefully in a different way. Each version then submits its answer to voter or decider which determines the correct answer, and returns that as the result of the module. This system can overcome the design faults present in most software by relying upon the design diversity concept.

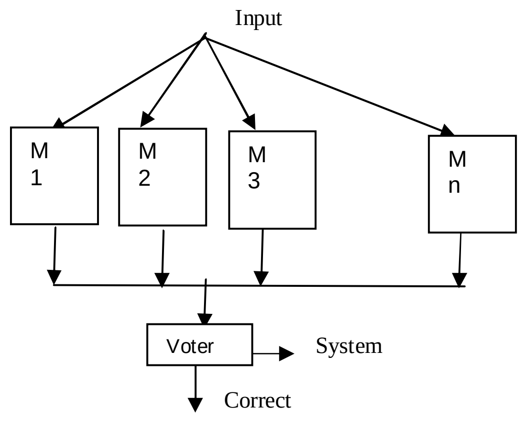

Figure 3.1: N-version fault tolerant software model

Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. The voter produces an output if at least *M* out of *N* outputs agree (it is presumed that the probability that *M* wrong outputs agree is negligibly small). Otherwise, the system fails. Usually majority voting is used in which *N* is odd and *M* = (*N*+1)/2. N-version programming consists of an adjudication module called a voter, and n independently developed software versions (M1, M2, M3,é Mn), which are functionally equivalent. This NVP model is based on the same concepts as N-modular redundancy (NMR), which is a hardware fault-tolerant architecture. In the NVP model, all n software versions are executed for the same task at the same time (i.e., in parallel), and their outputs are collected and evaluated by the voter as shown in Figure 3.1.

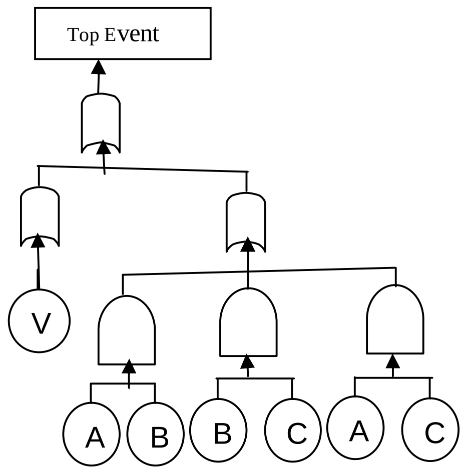

Figure 3.2: Fault tree diagram of 3-version FTS

The majority of the outputs determine the voter (V) decision. For the ease of computation of failure probability, n-version fault tolerant software (FTS) model can be converted into fault tree diagram. A fault tree equivalent of 3-version FTS (2 out-of-3) is shown in Figure 3.2.

3.2. Recovery block architecture

Another approach to software fault tolerance is the ñRecovery Blockò. As the name implies, the basic goal is to detect a software fault in a program, recover the machine state at the time the faulty program was entered, and execute next version that performs the same function as the faulty program. A number of independently designed programs that perform the same function are developed. The adjudicator (acceptance test) is the component which determines the correctness of the various blocks to try.

If an acceptance test detects erroneous output, then the next one is executed, etc., until an acceptable output is obtained. If all versions are deemed faulty, an error is posted. The Figure 3.3 illustrates the technique.

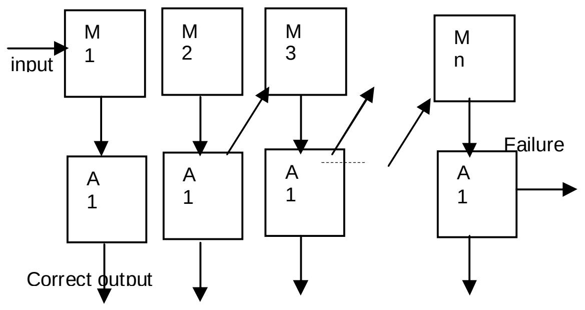

Figure 3.3: Recovery block

3.3. N shelf-checking programming (NSCP).

Self-checking software are the extra checks, often including some amount of check pointing and rollback recovery methods added into fault-tolerant or safety critical systems. In NSCP, N modules are executed in pairs. The outputs from the modules are compared and then the outputs of each pair are tested and if they do not agree with each other, the response of the pair is discarded. The technique is shown in Figure 3.4 for N=4. If a comparison of the outputs of the first pair of modules, M1 and M2, is successful, then the output is passed to the next phase of computation and system is successful. If these outputs disagree, then a comparison of the outputs of the second pair of modules, M3 and M4, is made. If the outputs of the second pair are agreed, then the output is passed to the next phase. Otherwise the system fails.

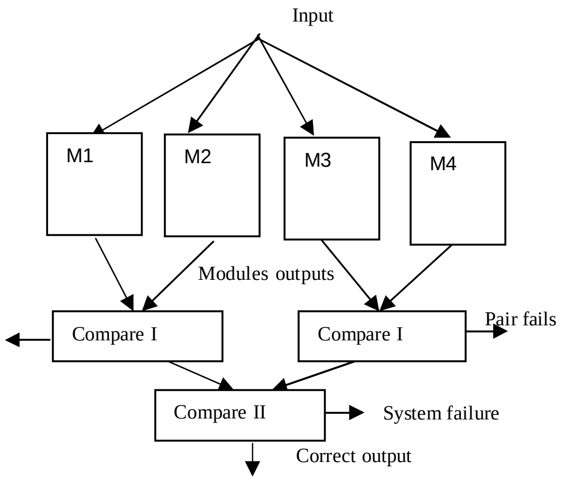

Figure 3. 4: N self-checking programming

3.4. Adjudication by voting

Majority voting:

Majority voting: In m-out-of-n fault tolerant software system, the number of version is N, and m is the agreement number, or the number of matching outputs which the adjudication algorithm requires for the system success. The value of n is rarely larger than 3. In general, in majority voting, $m = \lceil (N+1)/2 \rceil$ where, m is ceiling function of *(N+1)/2*.

4. Problem statements

Consider a software system consisting of n module/component, where some of components are used in redundant fashion to make the system fault tolerant. The goal is to assign failure probability requirements to the n modules (versions), such that the pre-speciýed reliability requirements of the system are satisfied, at the minimal cost. It is assumed that failures of versions for each component are statistically independent and having no coincident failure and hardware faults has not been taken into account. All the functionally equivalent module consist different number of faults depending on size (KLOC), but fault density is same

4.1. Model formulation

Two system failure probability allocation techniques are used: first one is, cut set method based on software Fault tree analysis (SFTA) [7], and second allocation technique, which is based on minimization of total cost (testing time/effort) and taking failure probability expression (obtained from fault tree or event tree) as constraints.

4.1.1 Cut set method: this method [7] analyses the logic relationships among the components (or applications) of the software, which may cause the root event to occur. A cut set of basic events whose occurrence causes the system to fail. A minimum cut set of a fault tree gives a minimum set of successful events necessary to satisfy the root. Presume that the maximum acceptable Failure probability of software system is *F*, and the system consists of *n* components $m_1, m_2, m_3, \ldots m_n$. By using SFTA, we get x minimum cut sets. If one minimum cut set contains ***i*** modules, then the maximum FR of each component in this minimum cut set is

$$F_{mj} \le \left(\frac{F}{x} \right)^{1/i}$$

Where,$j = 1, 2, 3 \ldots n$. (1)

If there exist intersections in the minimum cut sets, that is to say, the result of $F_{mj,}$ F may have ***k*** different values, then the minimum of them is taken as the value of *F.* This algorithm is a geometric mean algorithm in some sense, which is the reverse process of the traditional analysis of software failure rate by using SFTA. Cost minimization allocation model: The testing time (cost) is taken as objective function and failure probability expression, derived from equivalent fault tree (for n-version programming) and event tree [12] (for recovery block), is taken as constraint for reliability allocation model. There, two software reliability models are used to derive the objective function (testing time expression): Musa basic execution model and Musa-Okumoto logarithmic model. The problem is formulated as a nonlinear programming problem as follows

Musa basic execution time model:
Minimize **(2)**

$$C = \sum_{i=1}^{n} \frac{1}{\beta_i} \ln\left(\frac{\lambda_{0i}}{\lambda_i}\right)$$

Subject to $f(\lambda_1, \lambda_2, ...\lambda_n) \leq F$ (3)

where, C is testing time

λ_i (*failure intensity function of* i^{th} *module*) = $\lambda_{0i} \times \exp[-\beta_i C]$

β_i *is rate of failure decreament*

λ_{0i} is initial failure intensity of i^{th} module
F is goal failure probability of software system
The numerical values given below are either experimental or assumed [1].

$$\lambda_{0i} = f \times K \times \omega_{0i} = r \times K \times \rho_{0i} / Q = 50.4(failure / hr)$$

$$\beta_i = B \cdot \frac{\lambda_{0i}}{\omega_{0i}} = \frac{r \times K}{I_s(SLOC)} = \frac{10.5}{I_s}$$

f is execution frequency
Q (is code expansion ratio) = 4.5
ω_{0i} (Inherent faults) = $\rho_{0i} \times I_s$

r (Execution speed of processor) = 25 MIPS
K (Fault exposure ratio) = 4.2×10^{-7}
B (Fault reduction efficiency factor)= 0.85.(from Table 2.1.)
ρ_{0i} (Fault density) = 6 faults /KLOC (taken a average value)

Musa-Okumoto logarithmic model:
Testing time is given by

Objective.function

$$\tau = \frac{\beta_0}{\lambda_0}\left(\frac{\lambda_0}{\lambda(t)} - 1\right)$$

$$Cost = \sum_{i=1}^{n} \frac{I_{si} D_{\min}}{\lambda_{0i}}\left(\frac{\lambda_{0i}}{\lambda_i} - 1\right)$$ é .(4)

Constraints: $f(\lambda_1, \lambda_2, ...\lambda_n) \leq F$ (from eq. (3)).
Where, n is number of components,
D_{min}=4 faults per KLOC [14],
I_{si}= source lines of code of i^{th} module
ə$_i$= failure intensity function of i^{th} module
ə$_{0i}$= initial failure intensity function of i^{th} module

4.2. Reliability allocation of N-version programming architecture.

Consider a fault tolerant system consisting only one application as shown in Figure 4.1, where the modules M1, M2, and M3 are executed concurrently and 2-out of 3, is required to run the system successfully. The module *V* compares and checks the output of the modules whether it should be accepted or not. For the ease of computation, the 3-version software fault tolerant model shown in Figure 4.1 is converted to an equivalent fault tree diagram, as shown in Figure 4.2. Where, the modules *A, B,* and *C* (failure probability of M1, M2 and M3 respectively) are executed concurrently and at least 2-out of them is required to run the system successfully. The module *V* (failure probability of voter) checks for the most appropriate result out of the outputs of all the three versions.

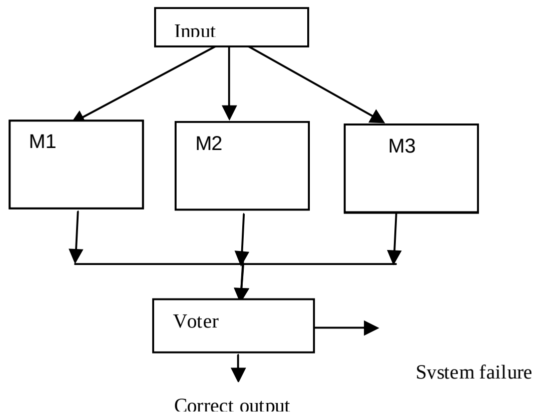

Figure 4.1: 3-version fault tolerant software model

Failure rate allocation using SFTA:
Minimal cut sets of the fault tree shown in Figure 4.2 are V (voter), AB, AC, BC. Let failure rate of the software (top event) = 0.03. F_v is failure rate of module V, Then by using Eq. (1), we get:

$$F_V = \left(\frac{F}{4}\right)^{1/1} = 0.0075$$

$$F_A = F_B = F_C = \left(\frac{F}{4}\right)^{1/2} = 0.0866$$

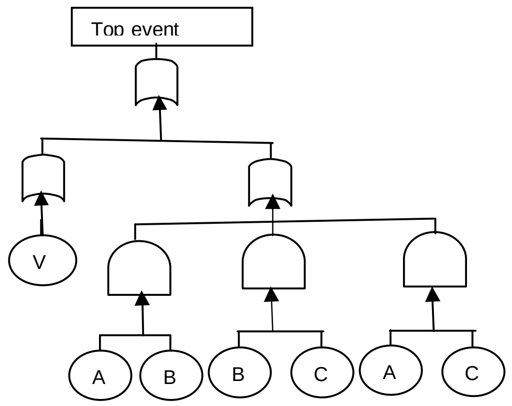

Figure 4.2: Fault tree equivalent to 3-version fault tolerant software model

Allocation using cost minimization
Let the goal failure rate of software is 0.03
The objective function is given (derived from (Musa basic execution model)Eq. (2)) by.

$$Cost = \frac{I_V}{10.5}\ln\left(\frac{50.4}{V}\right) + \frac{I_A}{10.5}\ln\left(\frac{50.4}{A}\right) + \frac{I_B}{10.5}\ln\left(\frac{50.4}{B}\right) + \frac{I_C}{10.5}\ln\left(\frac{50.4}{C}\right)$$

The objective function is given (derived from(Musa Okumoto Logarithmic model) Eq. (4)) by.

$$Cost = \frac{I_V \times 4}{50.4}\left(\frac{50.4}{V} - 1\right) + \frac{I_V \times 4}{50.4}\left(\frac{50.4}{A} - 1\right) +$$
$$\frac{I_V \times 4}{50.4}\left(\frac{50.4}{B} - 1\right) + \frac{I_V \times 4}{50.4}\left(\frac{50.4}{C} - 1\right)$$

The constraint is given by:

$$V + \overline{V}AB + \overline{V}\,\overline{A}BC + \overline{V}\,\overline{B}AC \leq 0.03$$

This constraint is derived from fault tree shown in Figure 4.2.
Other constraints can be considered as:
V>0.001; V<0.999;
A>0.001; A<0.999;
B>0.001; B<0.999;
C>0.001; C<0.999;

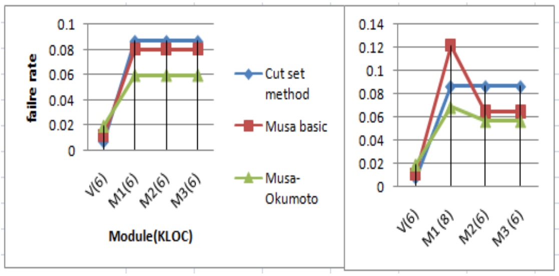

Figure 4.3: Failure rate vs. module size, in 3-version programming architecture

After solving the equations (objective function and constraints), the failure rate allocated to different modules (result of all three, cutest, Musa basic execution, and Musa Okumoto logarithmic methods) is plotted and are shown in Figure 4.3. From the graph shown in Figure 4.3, we can infer that the module v is supposed to be more reliable (least failure rate) and the allocated failure probability of modules A, B and C may be different based on their size (i.e. complexity). If modules sizes are equal then the allocated values are identical to estimated by using SFTA (cut set method). Musa-Okumoto model giving more appropriate allocation among three approaches (tabulated above), because, as size differs slightly, allocated value also differs slightly, but Musa basic execution model gives more variation where as cut set method gives no variation.

4.3. Reliability allocation for recovery blocks architecture.

In recovery block architecture, modules (versions) are not in n-modular redundancy fashion exactly, but it can be considered in sequential and standby fashion, so rather than converting this into an equivalent fault tree diagram, forming an event tree is more appropriate. Recovery block architecture of two modules (M1, and M2), and two adjudicator (A1 and A2) is shown in Figure 4.4.
An equivalent event tree diagram of Figure 4.4 is shown in Figure 4.5

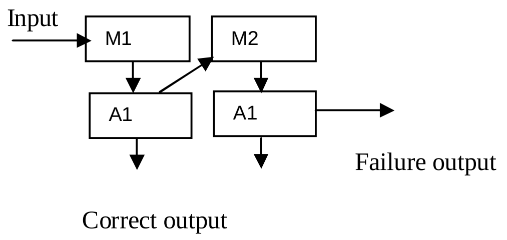

Figure 4.4: Recovery block architecture of 2-modules

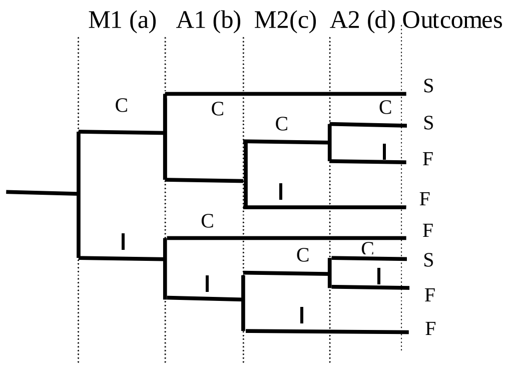

Figure 4.5 : Event tree of a 2-module recovery block.

Where,
C stands for correct output,
I stand for incorrect output,
S stands for success,
F stands for failure,
and a, b, c, d are failure probability M1, A1, M2, and A2 respectively.
Probability of failure of the event tree (shown in Figure 4.4) is given by

$$\overline{a}\,b\,\overline{c}\,d + \overline{a}bc + a\overline{b} + abd\,\overline{c} + abc$$
$$= a\overline{b} + bc + \overline{c}bd$$

The objective function is derived from Eq. (2) and given by (From Musa basic execution model)

$$C = \frac{I_{M1}}{10.5}\ln\left(\frac{50.4}{a}\right) + \frac{I_{M2}}{10.5}\ln\left(\frac{50.4}{c}\right) +$$
$$\frac{I_{A1}}{10\quad 5}\ln\left(\frac{50.4}{b}\right) + \frac{I_{A2}}{10.5}\ln\left(\frac{50.4}{d}\right)$$

The objective function is derived from Eq. (4) and given by (From Musa-Okumoto logarithmic model)

$$Cost = \frac{I_{M1} \times 4}{50.4}\left(\frac{50.4}{a} - 1\right) + \frac{I_{M2} \times 4}{50.4}\left(\frac{50.4}{c} - 1\right) + \frac{I_{A1} \times 4}{50.4}\left(\frac{50.4}{b} - 1\right) + \frac{I_{A2} \times 4}{50.4}\left(\frac{50.4}{d} - 1\right)$$

Constraint is derived (from event tree) as.

$$a\overline{b} + bc + \overline{c}bd <= 0.03$$

Other constraints are: a>0.001; a<0.999;
b>0.001; b<0.999;
c>0.001; c<0.999;
d>0.001; d<0.999;

Because failure rate beyond the range (0.001,0.999) is not practical.

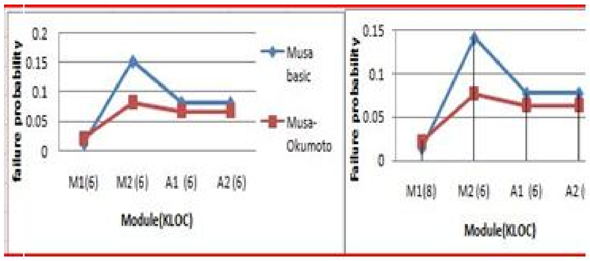

Figure 4.6: failure probability vs. module size plot of 2-module recovery block

After solving the equations (objective function and constraints), the failure rate allocated to different modules(size) is plotted and are shown in Figure 4.6, and so we can see that first executable module (version) is supposed to be more reliable than the next one, which justify the fact that, in recovery block architecture the best version executes first then another best one and so on. For the ease of solving nonlinear programming, the failure rates of acceptance tests are taken equal. The results of allocation also give the idea to organize the sequence (order) of module (versions) to execute. The allocated value obtained from Musa basic execution model shows more difference in failure rate of first executable version than the next one, whereas, Musa-Okomoto shows less difference. Moreover, Musa-Okumoto model allocates better reliability(less failure rate) to all other modules at the cost of small increase in failure rate of first executable version than Musa basic model. From the allocated value of failure probability, many inferences can be taken out, e.g. how to allocate testing effort and other resources among the modules

4.4. N self-checking programming architecture

The N self-checking programming architecture as shown in Figure 3.4, can be broken into two part, (a) compare II is taken as acceptance test and compare I along with modules connected with it, is taken as module (version), and (b) compareI is taken as voter and modules connected with this are taken as versions. So, reliability allocation also can be done in two steps, for part (I) recovery block allocation can be applied, and for part (II) N-version programming allocation can be applied, which are discussed already.

5. Conclusion

Reliability allocation is a useful tool at design stage of software development process. Fault tree diagram and event tree diagram are used to make software system simpler for reliability allocation work, especially for Fault tolerant system. In recovery block architecture, not all modules (functionally equivalent version of software module) results in output, so it is not reasonable to consider all modules as N-modular redundancy or in series fashion. Basic approach of reliability allocation using fault tree has been discussed along with reliability allocation cost (testing time) minimization. Voter is considered more critical hence, allocated more reliability than versions (software module with similar functionality). Musa-Okumoto logarithmic model allocates better reliability to all other modules at the cost of small increase in failure probability of first executable version.

Scope and future work

(I) Many other software models can be used for the allocation process.

(II) Number of faults corrected or residual can be taken as objective function.

(III) Some other factors like utility and occurrence probability can be incorporated in reliability allocation constraints.

6. References

[1]. Peter B. Lakey, McDonnell Douglas Corporation, St. Louis, MO Ann Marie Neufelder, SoftRel, Hebron, KY, “SYSTEM AND SOFTWARE RELIABILITY ASSURANCE NOTEBOOK ñ, Section 6 &7, Produced by Rome Laboratory

[2]. Yashwant K. Malaiya and Jason Denton, ñWhat Do the Software Reliability Growth Model Parameters Represent?, Computer Science Dept. Colorado State University Fort Collins, CO 80523 , pp: 124 - 135 , 1997.

[3]. Yashwant K. Malaiya, ñReliability Allocationò, Computer Science Dept , Colorado State University Fort Collins CO 80523 USA.

[4]. Fatemeh Zahedi and Noushin Ashrafi, ” Software Reliability Allocation Based on Structure, Utility, Price, and Costò, vol. 17, no. 4, April 1991.

[5]. Yashwant E, òFault Exposure Ratio Estimation and Applicationsò, Li Naixin Microsoft Corp, Malaiya Computer Science Dept, 1996.

[6]. Michael R. Lyu Sampath Rangarajan, Aad P. A. van Moorsel, “Optimization of Reliability Allocation and Testing Schedule for Software Systemsò, Bell Laboratories, Lucent

Technologies 600 Mountain Avenue, Murray Hill, NJ 07974.

[7]. Jianwen Xiang, Kokichi Futatsugi, ñFault Tree Analysis of Software Reliability Allocationò, School of Information Science, Ishikawa, 923-1292 Japan.

[8] R. Keithscott, James W. Gault, and David F.Mcallister, ñFault-Tolerant Software Reliability Modelingò, vol.se-13, no.5, May 1987

[9]. Longstreet, D., ñFunction Points Step by Stepò, Longstreet Consulting, Inc., January 1999.

[10] Rani, Misra R.B,òEconomic Allocation of Target Reliability in Modular Software Systems ñ, RAMS 2005.

[11].J.B.Fussell, E.F. Aber, R.G.Rahl, ñOn the Quantitative Analysis of Priority AND Failure Logicò, IEEE Transactions on Reliability, vol. R-25, no.5, December 1976.

[12]. Keith Scott, James W. Gault, and David F.Mcallister, ñ Fault-Tolerant software Reliability Modeling IEEE Transactions on Software Engineering, vol. se-13, no.5, may 1987

[13]. D.F.McAllister and M.A.Vouk, ñFault-Tolerant Software reliability Engineeringò, North Caroline state University, Section 14.

[14]. Michael Grottke ñSoftware Reliability Model Studyò, IST-1999-55017 Deliverable A.2

[15]. Michael R. Lyu, Sampath Rangarajan, ñOptimal Allocation of Test Resources for
Software Reliability Growth Modeling in
Software Developmentò, IEEE Transactions
On Reliability, Vol. 51, No. 2, June 2002

[16]. Yuri Goegebeur ñThe Poisson and the non homogeneous Poisson processò Department of Statistics, University of Southern Denmark, 05-Statistical Simulation, December 3, 2007