ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/264090326

A study of N–version programming and its impact on software availability

Article *in* International Journal of Systems Science · October 2014

DOI: 10.1080/00207721.2013.763299

CITATIONS
12

READS
2,291

3 authors, including:

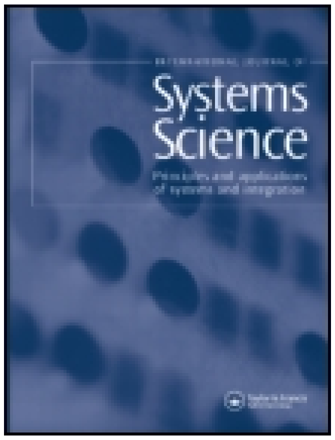
Chengjie Xiong
China Star Optoelectronics Technology, Inc., China, Shenzhen
3 PUBLICATIONS **25** CITATIONS

SEE PROFILE

All content following this page was uploaded by Min Xie on 18 August 2015.

The user has requested enhancement of the downloaded file.

This article was downloaded by: [NUS National University of Singapore]
On: 29 January 2015, At: 03:13
Publisher: Taylor & Francis
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK



International Journal of Systems Science
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tsys20

A study of N-version programming and its impact on software availability

Min Xie[ac], Chengjie Xiong[b] & Szu-Hui Ng[c]
[a] Department of Systems Engineering and Engineering Management, City University of Hong Kong, Hong Kong SAR
[b] Shenzhen China Star Optoelectronics Technology Co. Ltd., Shenzhen, Guangdong, China
[c] Department of Industrial and Systems Engineering, National University of Singapore, Singapore
Published online: 23 Jan 2013.

Click for updates

To cite this article: Min Xie, Chengjie Xiong & Szu-Hui Ng (2014) A study of N-version programming and its impact on software availability, International Journal of Systems Science, 45:10, 2145-2157, DOI: 10.1080/00207721.2013.763299

To link to this article: http://dx.doi.org/10.1080/00207721.2013.763299

PLEASE SCROLL DOWN FOR ARTICLE

Taylor & Francis makes every effort to ensure the accuracy of all the information (the “Content”) contained in the publications on our platform. However, Taylor & Francis, our agents, and our licensors make no representations or warranties whatsoever as to the accuracy, completeness, or suitability for any purpose of the Content. Any opinions and views expressed in this publication are the opinions and views of the authors, and are not the views of or endorsed by Taylor & Francis. The accuracy of the Content should not be relied upon and should be independently verified with primary sources of information. Taylor and Francis shall not be liable for any losses, actions, claims, proceedings, demands, costs, expenses, damages, and other liabilities whatsoever or howsoever caused arising directly or indirectly in connection with, in relation to or arising out of the use of the Content.

This article may be used for research, teaching, and private study purposes. Any substantial or systematic reproduction, redistribution, reselling, loan, sub-licensing, systematic supply, or distribution in any form to anyone is expressly forbidden. Terms & Conditions of access and use can be found at http://www.tandfonline.com/page/terms-and-conditions

International Journal of Systems Science, 2014
Vol. 45, No. 10, 2145–2157, http://dx.doi.org/10.1080/00207721.2013.763299

Taylor & Francis
Taylor & Francis Group

A study of N-version programming and its impact on software availability

Min Xie[a,c], Chengjie Xiong[b,*] and Szu-Hui Ng[c]

[a]*Department of Systems Engineering and Engineering Management, City University of Hong Kong, Hong Kong SAR;* [b]*Shenzhen China Star Optoelectronics Technology Co. Ltd., Shenzhen, Guangdong, China;* [c]*Department of Industrial and Systems Engineering, National University of Singapore, Singapore*

(*Received 11 May 2009; final version received 13 December 2012*)

N-version programming is a useful approach to improve the quality of software, especially for safety-critical systems. Positive performance in enhancing software availability is an expected result. In this paper, a software availability model for the study of the impact of N-version programming technique is proposed and investigated. The characteristics of the N-version software system and its operation and failure process are analysed. Based on this analysis, the time-dependent behaviour of the software system, which alternates between online and offline states, is described using a Markov chain. This model derives quantitative measures of software availability. Numerical examples and comparisons are also presented in this paper to directly illustrate N-version programming's positive impact on software availability measures. N-version programming generally provides a positive impact on the system. However, it does not always guarantee a higher availability performance. General recommendations are provided on N-version software structure design based on cost-effective criteria.

Keywords: N-version programming; software availability; Markov chain; optimal software structure

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

1. Introduction

Software failures in complex systems are troublesome for system users, and any malfunction in a safety-critical system may lead to direct, disastrous consequences. Since testing cannot eliminate all software faults (Lyu 1996) and intensive, thorough software testing is very expensive, an affordable method to improve the quality of complex systems is required. Software fault tolerance is such a method. There are primarily two types of fault tolerance techniques. One is the recovery block technique (Randell 1975; Berman and Kumar 1999; Mansour and Dillon 2011), and the other is the more popular N-version programming technique (Avizeinis 1985; Dugan and Lyu 1994; Dai, Xie, Poh, and Ng 2004; Wang 2012).

The N-version programming (NVP) technique is based mainly on redundancy. It involves the parallel execution of several independently developed programs that are functionally equivalent. Each single version behaves similarly to an ordinary software program, but the N-versions, as a whole, have better performance in many aspects.

The research on N-version programming received much effort and focus. In the early stages, researchers mainly focused on the modelling of N-version programming (Littlewood and Miller 1985; Nicola and Goyal 1990). These works discussed the possibility of simultaneous failures among different versions, and this topic later continued with further research on failure correlation in N-version programming software systems (Gutjahr 2001; Dai et al. 2004; Levitin and Xie 2006; Lin, Chang, and Fiodella 2012). In recent years, some literature has dealt with N-version programming problems under software reliability constraints. Some authors have focused on the cost (Bhaskar and Kumar 2006; Peng, Levitin, Xie, and Ng 2011) and optimal structure of N-version programming (Dai et al. 2004; Levitin 2005; Yamachia, Tsujimuraa, Kambayashia, and Yamamoto 2006).

Software availability is one of the most important properties of high-quality software systems. As the technology matures, so does the demand on software availability (Tokuno and Yamada 2003; Xiong, Xie, and Ng 2011; Tokuno 2012), especially with the increased use of more complex systems. Even a very small portion of software downtime could lead to a significant loss of revenue. For example, if the inter-bank clearance system malfunctioned even if only for several minutes, millions of dollars could be lost.

The problem of software availability derives from software-repairing activities. When software failures occur, repair services are usually necessary. Depending on the occasions and phases of the software application, repair services may present in the form of a team of professional software maintenance personnel, a software patch, a new distribution of software, etc. Regardless of the type of repair utilised, all of them take time to complete the required work.

*Corresponding author. Email: xiongchengjie@tcl.com

© 2013 Taylor & Francis

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

In early software availability studies, researchers had put more effort in the mathematical modelling work. Later, some authors studied the impact of imperfect debugging on software availability (Lee, Lee, and Park 2005; Peng et al. 2011). Recently, there has been more emphasis on user-oriented software availability assessments (Tokuno and Yamada 2007; Xiong et al. 2011) and the availability of specific systems (Ababneh 2008; Sergiy, Parnas, Mendiratta, and Murphy 2008).

Although availability of single-version software has been researched, there has been little effort on the investigation of availability impact from N-version programming. In this research, availability issues in N-version programming systems are considered. Al Markov's chain-based software availability model of N-version programming system is proposed. While the research in this area is still inadequate, the proposed model improves the understanding of the software availability issue, systematically, with better estimation accuracy. This model offers explicit consideration and analysis of the behaviours of each sub-version program. It makes the software process more apparent and understandable for users and may serve as a decision-making tool for software practitioners who seek building robust software systems with high availability and are under budget restrictions.

This paper is organised as follows. In Section 2, the N-version programming software system is described briefly. Then, with stated assumptions, the availability model is built. In Section 3, interactions among different sub-versions of a 2-out-of-3 N-version programming system are simulated and their impact on the system's availability is analysed. A comparison with traditional single-version software is also presented in this section. Section 4 presents a numerical example of applying the proposed model in deciding how to construct a software system under availability and budget restrictions. Summaries and conclusions are provided in Section 5.

2. N-version programming software availability modelling

2.1. N-version programming systems

Traditional software systems have only one version. Once data are input into the software, they are processed homogenously under the same path all the time. Any fault in the path would result in errors in the output. To reduce this problem, software engineers need to eliminate as many faults in the version as possible. However, it is difficult, if not impossible, to clear all faults in the path, so the concept of fault-tolerant software was introduced, as a form of software redundancy. Users expect better reliability/availability performance of software systems since its inception. The incorporation of redundancy or fault tolerance into software systems enhances performance levels.

One of the best-known fault-tolerant software design methods is N-version programming. The redundancy created by functionally equivalent but independently developed software modules (called versions) is the basis for this method. It has received considerable attention from software practitioners and researchers (Avizeinis 1985; Chatterjee, Misra, and Alam 2004; Xie, Dai, Poh, and Lai 2004; Wattanapongskorn and Coit 2007), especially in safety-critical systems, such as airport control systems and missile navigation systems. Despite criticism, such as trade-offs between additional computing resources and software performance (Gutjar and Uchida 2002; Wattanapongskorn and Levitan 2004), N-version programming outperforms traditional software design methods from the view of reliability consideration (Teng and Pham 2002). With fast-advancing hardware technology, computing resources are no longer the key constraint for software performance, and N-version programming is becoming more widely applied in different software systems due to the increasing demand of high-performance software.

Just as its name indicates, an N-version programming software system usually consists of two or more structurally different, but functionally equivalent, versions (Avizeinis 1985). Different versions of software process input data separately and simultaneously. Versions work separately and collaborate under a voting mechanism so that failures within minority versions will not affect the final output. In the voting mechanism, the system simply collects the simple majority as the final output of the whole system. In all, an N-version programming software system is a software system that utilises redundancy to reduce the probability of encountering failure. Redundancy has been the focus of much of the recent research (Maciejewski and Caban 2008; Ouzineb, Nourelfath, and Gendreau 2008; Taboada, Espiritu, and Coit 2008). However, redundancy in fault-tolerant software is somehow different, and it is a repairable system (Zhang 2008). A simple comparison between traditional software and N-version programming software is illustrated in Figure 1.

Ideally, the different versions within the N-version software system should be independent to reduce the possibility of failure triggered by the same input. Some authors call such input dependency, between different versions, failure correlation (Dai et al. 2004; Levitin and Xie 2006). However, this ideal case cannot be guaranteed, and different versions tend to be correlated to some extent, albeit slightly (Teng and Pham 2002). Some authors (Knight and Leveson 1986; Pham 1992) experimentally revealed that even though different teams developed different versions independently, they do not necessarily fail independently. Laprie, Arlat, Beounces, and Kanoun (1990) classified faults among different version as either independent faults or related common faults, and identified that common faults usually accounted for only a very small portion of all the faults. In this paper, only the ideal case that all versions work and cooperate independently is taken into consideration.

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

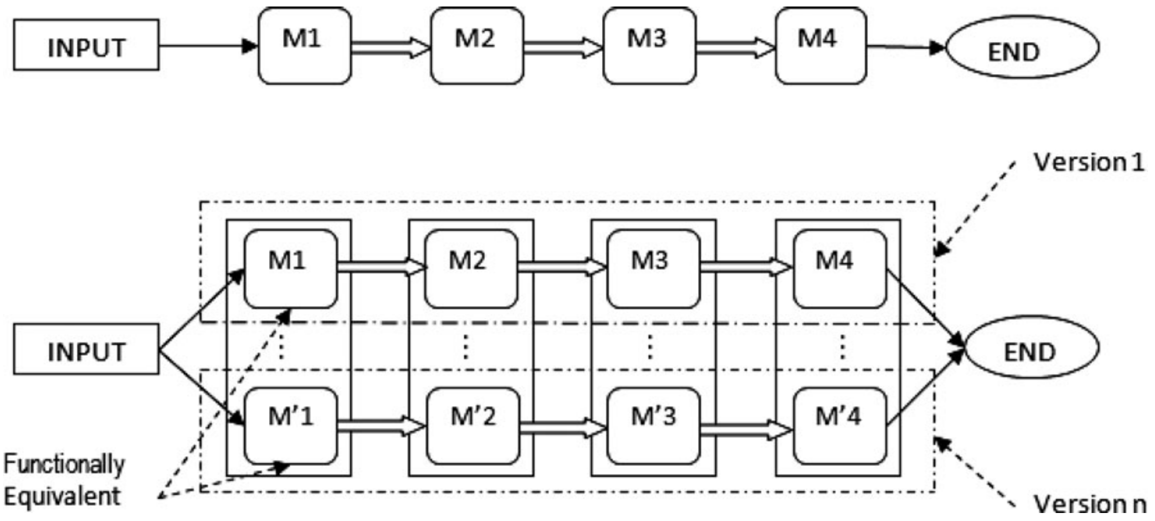


Figure 1. Traditional software versus N-version programming software.

2.2. Proposed model of software availability

While the literature in software reliability is vast and intensive (Xie 1991; Lyu 1996), few study the problem of software availability. The main difference between software reliability and software availability is that software reliability focuses on the software failure process, while software availability is based on both the software failure and the software maintenance process (Xiong et al. 2011). Some authors argue that software reliability is a metric for professionals, while software availability is more relevant for end-users (Tokuno and Yamada 2003).

2.2.1. Preliminaries

N-version programming techniques are often applied to large software systems in which statistical deduction is meaningful. Following are the four assumptions:

Assumptions:

(1) *An N-version programming software system consists of several different and independent sub-versions. Data are processed simultaneously on all versions.*
(2) *Each of the sub-versions has a number of initial faults that can trigger failures in each corresponding path in the operation. Faults in different versions are not correlated.*
(3) *When a failure occurs in a sub-version, this version stops working and will be restored. Failure restoration takes time and effort. Work in other versions is not compromised, and restored faults do not recur.*
(4) *When more than half of all of the sub-versions of the paths fail, the software system performance is a failure.*

The following notations are used in this paper:

Notations:

n:	total number of sub-versions in the N-version programming system
N_i:	number of initial faults in sub-version *i*
$\lambda_{i,j}$:	failure rate of sub-version *i* when *j* failures have been encountered
$\mu_{i,j}$:	restoration rate of sub-version *i* when *j* failures have been encountered
$p^i_{(j,0)(k,0)}$:	probability that version *i* will be in working state at time *t* after *k* faults are to be removed, given that the software is currently in the working state and *j* faults have been removed previously
(j, w):	duplet that represents a sub-version state
$A^i(t)$:	instant software availability of sub-version i at time *t*
$A(t)$:	instant software availability at time *t*
$\bar{A}^i(t)$:	average software availability over the period $(0, t]$
D_i:	initial hazard rate of sub-version *i*
E_i:	initial failure restoration rate of sub-version *i*
C_{total}:	total cost during the software operational cycle
C_o:	cost of software operation
C_m:	cost of software maintenance
C_r:	risk cost if the software system is unavailable
c_o:	expected unit time cost of software operation
c_i:	expected unit time cost of maintenance cost of sub-version *i*
c_r:	expected unit time cost if the software system is unavailable
L:	expected length of software operation cycle
A_{req}:	minimal requirement of software availability

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

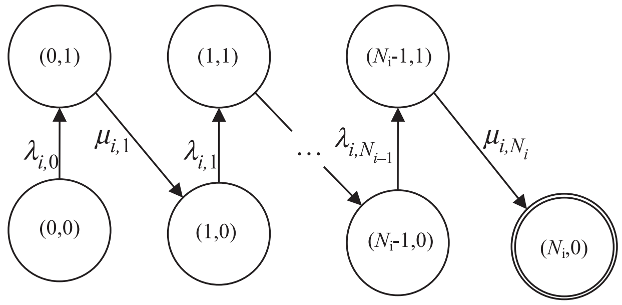


Figure 2. State transition diagram of sub-version *i*.

2.2.2. Mathematical Formulation

All *n* (*n* is usually odd) sub-versions are functionally equivalent, so it is reasonable to believe that all *n* versions should behave similarly. The state of a single sub-version i can be denoted using a duplet (*j*, *w*), where $j = 0, 1, \ldots, N_i$ is the total number of failures encountered in version i up to time *t* and $w = \begin{cases} 0 \\ 1 \end{cases}$ indicates this version is working (0) or non-working (1) in restoration. For a single sub-version, it alternates between up and down stages. If one can explicitly analyse the behaviours of a single sub-version of N-version programming software, one can conclude a whole image of the N-version system. State transition diagram of a single sub-version i can be illustrated using the graph below (Figure 2):

Consider a stochastic process, $\{X_i(t), t \geq 0\}$ and $X_i(t) = (j, w)$. The process that is presented in Figure 1 can be modelled as a pure-birth Markov chain. For version *i*, the time until the next failure is exponentially distributed with mean time $1/\lambda_{i,j}$, while the time until the next restoration is exponentially distributed with mean time $1/\mu_{i,j}$. Restorations can be done either by professionals or by a self-checking mechanism.

A further assumption is that no failures would occur in any version at the beginning of an operation. Software availability is defined as the probability, at a given time in a given environment, that the software system is working and accessible. According to the above assumptions, software availability is the probability that the software system is functional. In other words, the availability of N-version programming software is the probability that no more than half of its sub-versions are down. Since w_i can be used to indicated that each sub-version is either working (0) or non-working (1), summing up w_i produces a rough picture of the status of the whole system. Denoted as $A(t)$, the software availability of an N-version programming software can be expressed as:

$$A(t) = \Pr\left\{ \sum_{i=1}^{n} w_i \leq \left\lfloor \frac{n}{2} \right\rfloor | X_i(t) = (j_i, w_i), i \quad = 1, 2, \ldots, n \right\}, \qquad (1)$$

where $\lfloor n/2 \rfloor$ is the floor function which returns the greatest integer that is no larger than $n/2$.

This availability $A(t)$ means that the software system is operational at time *t*, given that the system was operational at time 0. Furthermore, the average software availability over the period (0, *t*] can be calculated as

$$\bar{A}(t) = \frac{1}{t} \int_0^t A(s)ds, \qquad (2)$$

and the average software availability represents the average proportion of operating time during the time interval (0, *t*], given that the software is operational at time 0. In most cases, this average figure is more meaningful and more widely used. However, Equations (1) and (2) only give a conceptual demonstration of how software availability can be obtained. Further derivation is required if quantitative measurements of software availability using explicit expressions is needed.

Since each version works independently, the behaviours of a single version can be analysed first. The probability that sub-version *i* will be in a certain state after elapsed time *t* is of interest. Denote $p^i_{(j,0),(k,0)}(t)$ as the probability that sub-version *i* will be in a working state at time *t* after *k* faults are removed, given that the software is currently in a working state at time 0 and *j* faults have been removed previously, and can be written as:

$$p^i_{(j,0)(k,0)}(t) = \Pr\{X_i(t) = (k,0) | X_i(0) = (j,0)\}, \ j \leq k,$$

for $j = 0, 1, 2, \ldots, k$ and $k = 0, 1, 2, \ldots, N_i$. The availability of this sub-version *i* at time *t* can be calculated as:

$$A^i(t) = \sum_{j=0}^{N_i} p^i_{(0,0)(j,0)}(t).$$

According to the assumptions stated above, the whole software system is available only when no less than half of its sub-versions are functioning. If the availability of a single version can be obtained, the availability of the whole system can also be derived directly. It can be proved that (please refer to Appendix)

$$p^i_{(0,0)(k,0)}(t) = \left\{ \prod_{l=0}^{k-1} \lambda_{i,l}\mu_{i,l} \right\} \sum_{j=0}^{k} \left\{ \frac{A_j + \mu_{i,k}}{S_A(j,k)} \exp(A_j t) + \frac{B_j + \mu_{i,k}}{S_B(j,k)} \exp(B_j t) \right\}, k < N_i, \qquad (3)$$

$$p^i_{(0,0)(N_i,0)}(t) = 1 + \left\{ \prod_{l=0}^{N_i-1} \lambda_{i,l}\mu_{i,l} \right\} \sum_{j=0}^{N_i-1} \left\{ \frac{\exp(A_j t)}{A_j S_A(j,1)} + \frac{\exp(B_j t)}{B_j S_B(j,1)} \right\}_i, \qquad (4)$$

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

where

$$\begin{cases} S_A(j,k) = \left\{\prod_{l=0}^{k}(A_j - B_l\right\}\left\{\prod_{\substack{m=0\\m\neq j}}^{k}(A_j - A_m)\right\} \\ S_B(j,k) = \left\{\prod_{\substack{l=0\\l\neq j}}^{k}(B_j - B_l)\right\}\left\{\prod_{m=0}^{k}(B_j - A_m\right\} \end{cases}.$$

and $\begin{cases} A_j \\ B_j \end{cases}$ are two roots of the quadratic equation $x^2 + (\lambda_{i,j} + \mu_{i,j})x + \lambda_{i,j}\mu_{i,j} = 0$.

The general expression of availability of the N-version software system can be derived by adapting the working-state probability:

$$A(t) = \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \left\{ \sum_{j=1}^{C_n^i} \left[\varsigma_{j1}\varsigma_{j2} \right] \right\}, \qquad (5)$$

where

$$\begin{cases} \varsigma_{j1} \\ = \left[\left(1 - \sum_{k=0}^{N_{(i+1)j}} p_{(0,0)(k,0)}^{(i+1)j}(t)\right) \cdots \left(1 - \sum_{k=0}^{N_{nj}} p_{(0,0)(k,0)}^{nj}(t)\right)\right] \\ \varsigma_{j2} \\ = \left[\sum_{k=0}^{N_{1j}} p_{(0,0)(k,0)}^{1j}(t) \sum_{k=0}^{N_{2j}} p_{(0,0)(k,0)}^{2j}(t) \cdots \sum_{k=0}^{N_{ij}} p_{(0,0)(k,0)}^{ij}(t)\right] \end{cases},$$

where $p^{1j}, p^{2j}, \ldots, p^{ij}$ denotes a unique combination of i different versions out of n. Substituting the working-state probability by (3) and (4) into the above expression, the explicit expression for software availability can be obtained.

3. Impact on software availability

In this section, the availability of a 2-out-of-3 system via N-version programming software design is studied. Although illustrative, this section reveals the potential of N-version programming in enhancing software availability, and the conclusions can be generalised to any k-out-of-n N-version programming systems.

3.1. Failure and restoration rates for different versions of software

3.1.1. Empirical analysis of software operation and restoration time

The operation and restoration time for software systems is an important metric for software users. Many authors have researched this issue. According to a study by Tamai and Torimitsu (1992), the average lifetime for mainframe-level business software systems is about 9 years, and this figure increases as the size (measured in KLOC [thousands {kilo} of lines of code]) of the software increases and while only minor maintenance work accumulates until a thorough system upgrade.

As it is hard to get industrial field data of software operation and restoration for practical analysis, simulation analysis based on empirical basis becomes meaningful. Appropriate failure and restoration rates represent software operation and restoration time, upon which software availability can be assessed. In reality, it is possible for a real software process to differ from the simulated process, but this problem can be solved by replacing failure and restoration rates that better model the process. Hence, the simulation analysis can be applied rationally in practice with minor changes.

3.1.2. Mathematical representation of software failure and restoration rates

The failure rate of a software system would gradually decrease with time, with the removal of more faults. Further, with time, the restoration rate would also decrease since the complexity (Nakagawa and Takenaka 1991) of future encountered faults increase.

Moranda (1979) proposed a geometric model from the view point that software reliability depends on debugging efforts, and earlier debugging activities have a larger impact on software reliability growth than later ones (Xie and Yang 2003). This model has been discussed and adopted by many researchers and is still popular with software practitioners (Dick, Bethel, and Kandel 2007). In this paper, for the sake of simplicity, Moranda's model describes the software failure-occurrence phenomenon. That is, when j faults have been corrected, the hazard rate is given by

$$\lambda_{i,j} = D_i k_i^j, \qquad 0 < k_i \leq 1,\ j \geq 0, \qquad (6)$$

where D_i is the initial hazard rate in sub-version i and k_i is a proportional factor. For the restoration rate, the assumption is similar to that of the hazard rate and takes the form of (Tokuno and Yamada 2003)

$$\mu_{i,j} = E_i r_i^j, \qquad 0 < r \leq 1,\ j \geq 0, \qquad (7)$$

where E_i is the initial failure restoration rate of sub-version i and r_i is a proportional factor. By substituting (6) and (7) into (5), an explicit software availability expression is obtained.

Note that Moranda's model is used to represent both failure and restoration rate merely as a matter of convenience. This proposed approach apparently allows for the use of other models. For real data, it is also important to test the validity of the used model. In real-life scenarios,

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

other approaches have been proposed for estimating the parameters in (6) and (7) with the historical data that were recorded during the development phase. Other useful metrics such as the number of remaining faults in the software can also be obtained under these approaches.

3.2. *N-version's impact on software availability: A simulation approach*

Interactions among sub-versions of an N-version programming system greatly influence system availability. Due to the uniqueness of different sub-versions, availability performance could vary if different sub-versions are chosen. Another aim of this research is discovering the possible availability behaviours of a 2-out-of-3 N-version programming system, considering different sub-version contents. The choice of the 2-out-of-3 N-version programming system is due to its simplicity of illustration, as well as its commonality in industrial cases. In the mean time, the results can also easily be expanded to any general k-out-of-n N-version programming systems.

A simulation approach is adopted in this research. Three different software versions are simulated. It is assumed that they are developed with different fault contents. Each version can perform the required tasks independently. Different software versions, as well as their collaboration, are simulated using different version parameters (failure/restoration rates) stated above. For the purpose of illustration, all parameters are pre-set with a reasonable value. For real applications, these parameters should come from field or experience data.

3.2.1. *Scenario 1*

Without loss of generality, for any programs, assume that the initial failure restoration rate is much higher than the initial hazard rate. What differentiates a well-developed program from a poorly developed one is the proportional factor of hazard rate and restoration rate. For a well-developed program, people expect that the failure restoration time is much shorter than the time to failure. When incorporating (6) and (7), such expectation can be explained by stating that the decreasing rate of failure restoration is slower than the decreasing of hazard rate. In mathematical terms, such an expectation can be explained as $k < r$.

Suppose that there are three well-developed sub-version programs. By incorporating the assumptions stated above (see Table 1 for details of each version), the initial failure restoration rate is 10 times that of the initial failure rate for all three sub-versions. The k and r values are set to meet the requirement of $k < r$ for all three sub-versions.

Table 1. Details of three versions in Scenario 1.

Version	$D_i; k$	$E_i; r$	k/r	No. of initial faults
A	0.10;0.80	1;0.90	0.89	10
B	0.10;0.85	1;0.95	0.89	10
C	0.10;0.70	1;0.95	0.74	10

Table 2. Average availability of different software systems in Scenario 1.

	Single-version			Three-version
Time (t)	A	B	C	ABC
1	0.96976	0.96974	0.96981	0.99473
2	0.95316	0.95309	0.95339	0.99043
10	0.92478	0.92442	0.92820	0.98250
30	0.92325	0.92280	0.93344	0.98346
50	0.92658	0.92627	0.94076	0.98530
70	0.92993	0.92984	0.94660	0.98679
100	0.93430	0.93443	0.95322	0.98840
120	0.93665	0.93652	0.95647	0.98896

For each version, note that the ratio $k/r < 1$. Some authors (Tokuno and Yamada 2003) argue that this ratio can serve as an index of software availability improvement. Comparisons are made among both different single-version and three-version programs. The values of average availability of different systems over time are shown in Table 2. Availability plots are shown in Figures 3 and 4.

It can be concluded that in this scenario, the performance of software availability of the three-version software system is better than any of its sub-version programs. For each single sub-version program, failure rate, restoration rate and the number of initial faults affect software availability. Availability is increased greatly if the system changes from single version to N-version, and the availability performance of N-version systems is smoother than that of their single sub-version counterparts, with no steep drop or increase in availability over time.

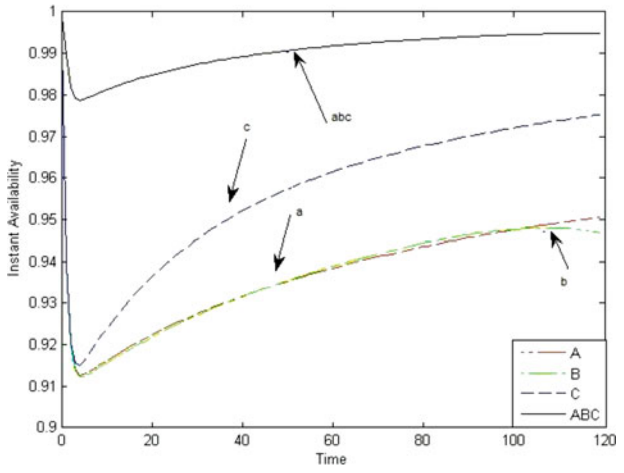


Figure 3. Scenario 1: instant availability.

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

Table 3. Details of three versions in Scenario 2.

Version	$D_i; k$	$E_i; r$	k/r	No. of initial faults
A	0.10;0.80	1;0.70	1.14	10
B	0.10;0.85	1;0.75	1.13	10
C	0.10;0.70	1;0.60	1.17	10

Table 4. Average availability of different software systems in Scenario 2.

	Single-version			Three-version
Time (t)	A	B	C	ABC
1	0.96976	0.96974	0.96981	0.99473
2	0.95316	0.95309	0.95339	0.99043
10	0.92478	0.92442	0.92820	0.98250
30	0.92325	0.92280	0.93344	0.98346
50	0.92658	0.92627	0.94076	0.98530
70	0.92993	0.92984	0.94660	0.98679
100	0.93430	0.93443	0.95322	0.98840
120	0.93665	0.93652	0.95647	0.98896

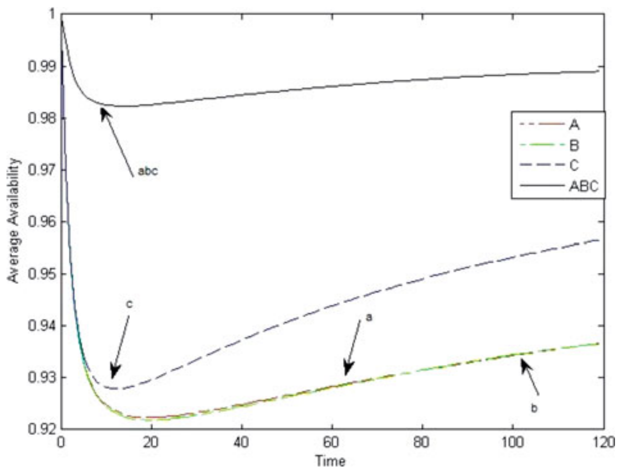


Figure 4. Scenario 1: average availability.

3.2.2. Scenario 2

Suppose that three poorly developed sub-version programs are provided. Details of each version are provided in Table 3. The initial failure restoration rate and the initial failure rate for all three sub-versions are adopted from Scenario 1, and the k and r values are set to meet the requirement of $k > r$ for all three sub-versions.

For this scenario, for each version, note that the ratio $k/r > 1$. Comparisons are made among both different single-version and three-version programs. The values of average availability of different systems over time are shown in Table 4. Availability plots are shown in Figures 5 and 6.

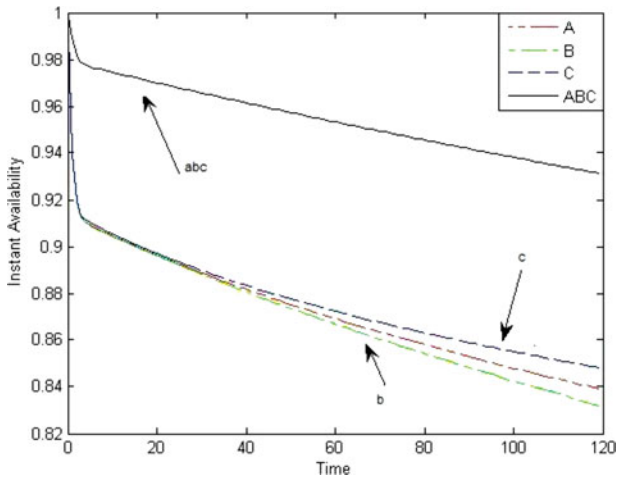


Figure 5. Scenario 2: instant availability.

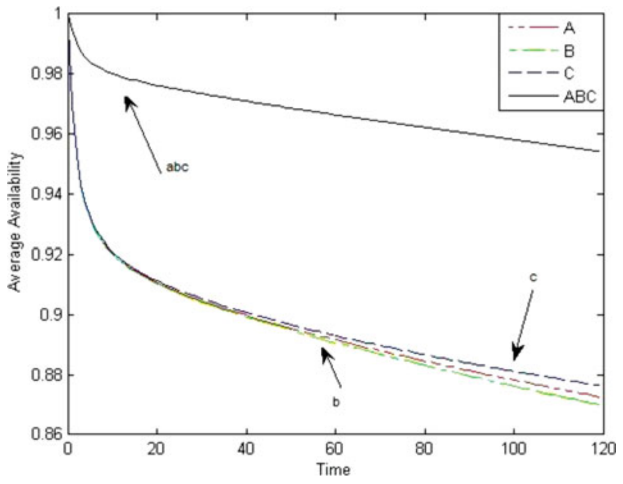


Figure 6. Scenario 2: average availability.

The same conclusion as in Scenario 1 can still be made. The performance of software availability of the three-version software system is better than any of the single sub-version programs. However, as all the ratios $k/r > 1$, the availability decreases with time. Actual industrial cases try to avoid this phenomenon.

3.2.3. Scenario 3

In this case, suppose that three sub-version programs are provided. One is well developed, while the other two are poorly developed. Details of each version are provided in Table 5. The initial failure restoration rate and the initial failure rate for all three sub-versions are adopted from Scenario 1.

Note that in this scenario, for Version A, the ratio is $k/r < 1$, while for Versions B and C, the ratio is $k/r > 1$. Comparisons are made among both different single-version and three-version programs. The values of

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

Table 5. Details of three versions in Scenario 3.

Version	$D_i; k$	$E_i; r$	k/r	No. of initial faults
A	0.10;0.80	1;0.90	0.89	10
B	0.10;0.85	1;0.50	1.7	10
C	0.10;0.70	1;0.50	1.4	10

Table 6. Average availability of different software systems in Scenario 3.

	Single-version			Three-version
Time (t)	A	B	C	ABC
1	0.96976	0.96969	0.96977	0.99473
2	0.95316	0.95275	0.95310	0.99036
10	0.92478	0.91187	0.91788	0.97899
30	0.92325	0.86492	0.89139	0.96743
50	0.92658	0.82241	0.87244	0.95650
70	0.92993	0.78205	0.85614	0.94525
100	0.93430	0.72633	0.83484	0.92815
120	0.93665	0.69425	0.82279	0.91739

average availability of different systems over time are shown in Table 6. Availability plots are shown in Figures 7 and 8.

In this scenario, the performance of software availability of the three-version software system no longer always outperforms its single sub-programs. In fact, the optimal strategy for this scenario is to construct the software system using a single version, A. Although the performance of the ABC collaboration is better at the beginning, A will outperform ABC as time increases. However, the performance of ABC collaboration is still much better than B and C alone.

Based on the above three different scenarios, it can be concluded that the N-version programming technique has very apparent, positive impact on a software system's availability performance. However, the availability performance of an N-version system is affected greatly by its sub-version programs' property and a larger N does not necessarily guarantee higher availability performance. Concluding from the above three scenarios, the strategy is to choose sub-versions with a ratio $k/r < 1$ to construct the N-version system. If the number of sub-versions with a ratio $k/r < 1$ is less than half of the required N-versions, the required N-version system with the highest availability can never be obtained and a better approach is to reduce the number of required versions. If all of the sub-versions are with a ratio $k/r > 1$, such versions are not appropriate for constructing software systems with availability constraints.

For software projects with a limited budget, availability is not a main concern, building a single version system with good reliability/availability should be the priority.

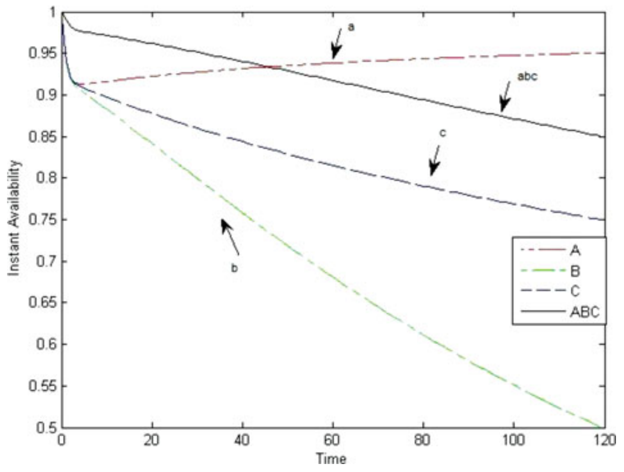


Figure 7. Scenario 3: instant availability.

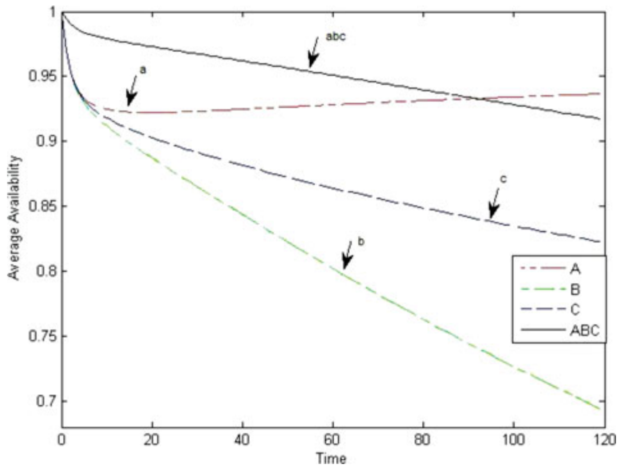


Figure 8. Scenario 3: average availability.

For systems with availability constraints, one also needs to look into the properties of different sub-versions. It is not a 'the-more-the-better' approach, as the interactions among different versions determine the final performance. The availability model proposed in this research provides a possible approach to establish the effects of such interactions.

4. Optimal software structure under availability and budget constraints

In this section, the model is applied to an optimal software structure (Levitin 2005) problem. Suppose that the structure of a software system is in question. The system can be either a traditional single-version system or a 2-out-of-3 N-version system. There is a strict requirement of software availability over the operational cycle, and total cost during the operational cycle needs to be reduced as much as possible. The software development cost is not considered. This is a typical problem that managers of IT (information technology) sectors often face.

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

4.1. Optimisation modelling

The total cost of an N-version programming software system during its operational cycle usually consists of three main components:

(1) Operation cost C_o: the cost incurred for normal software operation. If the expected length of operational cycle is denoted as L, the operation cost can be expressed as:

$$C_o = c_o L, \tag{8}$$

where c_o is the expected operation cost per unit time. This cost is commonly deterministic for most software systems.

(2) Maintenance cost C_m: the cost incurred for maintaining software system. Maintenance teams usually execute maintenance tasks, and different maintenance policies often correspond to the different team sizes. This cost is controllable by adapting different maintenance policies, and it can be expressed as:

$$C_m = \sum_{i=1}^{n} c_i L, \tag{9}$$

where c_i represents the current maintenance policy, and it is the expected maintenance cost per unit time for sub-version i. Normally, the initial failure restoration rate E_i of sub-version i is proportional to $c_i$and the relationship can be denoted as $E_i \propto c_i$. The most basic, initial failure restoration rate of the ith sub-version is denoted by E_{i0}, which corresponds with the cheapest maintenance policy. Without loss of generality, the policy can be written as:

$$\begin{cases} c_i = k_i \phi_i E_{i0} \\ E_i = k_i E_{i0} \end{cases} \quad k_i = 1, 2, 3, \ldots, \tag{10}$$

where $k_i$is the decision factor for the maintenance policy in version i and φ_i is a deterministic proportional factor. Generally, k_i can be interpreted as a multiple of the minimum of the maintenance team size. Then, the total maintenance cost can be expressed as:

$$C_m = \sum_{i=1}^{n} k_i \phi_i E_{i0} L; \quad k_i = 1, 2, 3, \ldots. \tag{11}$$

(3) Risk cost C_r: the cost incurred by an unavailable system. This can be expressed as:

$$C_r = \int_0^L c_r[1 - A(\tau)]d\tau, \tag{12}$$

where c_r is the expected cost per unit time if the system is unavailable.

(4) Development cost C_d: the cost incurred by developing the target software system. This is a fixed cost, and it is expected that the development cost of an N-version programming system is N times higher than the development cost of a single-version software system.

The total cost is the summation of the above four components. As we can see, the maintenance cost directly reflects the maintenance policy, and it also affects the software availability, thus indirectly affecting the risk cost. More maintenance efforts will lead to less risk cost, but they also increase the maintenance cost. Hence, an optimal maintenance policy minimises the total cost. If the availability requirement is quantitatively marked as A_{req}, the optimisation problem can be formulated.
Objective:

$$\min C_{\text{total}} = C_o + C_m + C_r + C_d = C_d + c_0 L + \sum_{i=1}^{n} k_i \phi_i E_{i0} L + \int_0^L c_r[1 - A(\tau)]d\tau, \tag{13}$$

subject to:

$$\begin{aligned} &A(\tau) \geq A_{req}, \forall \tau \in (0, L). \\ &k_i = 1, 2, 3, \ldots \\ &n = 1, 2, 3, \ldots \end{aligned}$$

The above optimisation problem has two decision variables, n and k_i. In the above, n is the major decision variable, which determines the structure of an N-version programming system and represents the number of sub-versions within the system; k_i represents the maintenance policy under a determined software structure. Once solved, the proposed model can find not only the optimal software structure but also the optimal maintenance policy under the optimal software structure.

4.2. A numerical example

To illustrate the proposed solution to the optimal software structure problem, the example in Section 3.2.1 is extended. Some cost factors (Table 7) are incorporated with the example in Section 3.2.1. The cost factors that are used here are just for demonstration purpose, and they could be replaced

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

Table 7. Maintenance cost factors.

Version	c_o	φ_i	E_{i0}	C_r	L	C_d
A	1	10	1	500	120	100
B	1	10	1	500	120	100
C	1	10	1	500	120	100

Table 8. Optimal software structure and maintenance policy.

A_{req}	n	k_i	C_{total}
0.95	1	$k_C = 2$	3942.4
	3	$k_A = 1; k_B = 1; k_C = 1$	4601.0
0.97	1	$k_C = 4$	5684.5
	3	$k_A = 1; k_B = 1; k_C = 1$	4601.0

by field data once acquired. With respect to maintenance policies, there is a choice between single-version software and three-version software. The aim here is to select the most appropriate software system with the lowest cost over the course of the operation cycle.

Depending on the requirement of software availability, two pairs of results will be illustrated by numerically solving the proposed model given by (13). The results are shown in Table 8, and the time value of cost is not considered.

Judging from the above results, we can see that if the availability requirement is 0.95, the optimal software structure is the single-version software of version C with maintenance policy $k_C = 2$; if the availability is 0.97, the optimal software structure is the three-version software with maintenance policy $k_A = 1; k_B = 1; k_C = 1$.

There is another interesting conclusion. If the cost of developing a software system is very high, one has to consider carefully the trade-off between choosing intensive software maintenance for a single-version software and a high-cost N-version programming system. For example, if it exceeds 642 in the above example, the optimal choice would always be the single-version software of version C, with an intensive maintenance policy.

Although this example is for illustrative purposes, the above results draw some meaningful conclusions. For systems with moderate availability requirements, N-version programming systems do not always serve as the most cost-effective method. The availability of traditional software systems can be enhanced to meet the availability requirement by putting more maintenance efforts at a lower cost. Many end-user software systems can be classified into this category, such as online enquiring systems. However, if the availability requirement is very high, the N-version programming systems will surely perform better in both availability and cost aspects. These conclusions can be generalised for any general N-version programming structure problems.

5. Conclusions

This paper presented a stochastic model to describe the software failure and restoration process, and it is useful for the evaluation of the impact on software availability when adopting an N-version programming technique. A Markov process model is used to describe the behaviour of a system alternating between operable and inoperable states, and explicit expressions of state probability are derived. Instantaneous and average software availability can be obtained based on these derivations. Numerical examples also show a firm support of N-version programming techniques, as it has a positive impact on software availability. The problem of an optimal software structure, considering availability and cost, is also covered and some general recommendations are also given.

This research could be extended by considering more realistic cost models or investigating release policies during the testing stage, as has been done in software reliability studies (Pham 2003; Huang and Lo 2006; Kapur, Bardhan, and Yadavalli 2007; Xie, Li, and Ng 2011; Li, Xie, and Ng 2012). An important assumption in this paper is the perfect fault removal and zero failure correlation, which cannot be guaranteed in a real-life practice. Further research incorporating imperfect debugging and failure correlation scenarios (Gutjahr 2001; Levitin and Xie 2006: Kapur, Pham, Anand, and Yadov 2011; Pievatolo, Ruggeri, and Soyer 2012) would be interesting. It is also worth analysing how many versions are needed to construct the most competent N-version programming software system under certain constraints.

Acknowledgements

The authors would like to thank the editor and referees for the suggestions on earlier versions of this paper. This research was partially supported by a research project grant (Science and Engineering Research Council [SERC], Singapore: grant number 072 1340050) and a grant from the City University of Hong Kong (project no. 9380058).

Notes on contributors

Min Xie received his PhD in Quality Technology in 1987 from Linkoping University in Sweden. He was awarded the prestigious Lee Kuan Yew (LKY) research fellowship and joined the National University of Singapore in 1991. Currently, he is with the City University of Hong Kong as Chair Professor of Industrial Engineering. Prof. Xie has authored or co-authored numerous papers and eight books on quality and reliability engineering, including *Statistical Models and Control Charts for High-Quality Processes* by Kluwer Academic, *Advanced QFD Applications* by ASQ Press, *Weibull Models* by Wiley and *Computing Systems Reliability* by Kluwer Academic. He is a Department Editor of *IIE Transactions,* Area Editor of *Computers & Industrial Engineering*, Associate Editor of *IEEE Transactions on Reliability*, and on the editorial board of over 10 other international journals. Prof. Xie has served as chair and committee members in over 100 international conferences and delivered keynote speeches at several of them. He has supervised over 30 PhD students. Prof. Xie is an elected fellow of IEEE.

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

Cheng-Jie Xiong received his BE degree in 2006 from Wuhan University, China, and his PhD degree in 2011 from the National University of Singapore, Singapore. He has published several journal papers about quantitative analysis of software quality issues. His major research interest includes software reliability, software availability, optimal resource allocation and probabilistic modelling. He is now working as an IE (industrial engineering) manager in a leading Chinese TFT-LCD (thin film transistor liquid crystal display) manufacturing company in Shenzhen, China, and his current work focuses on production resource allocation and optimal production scheduling.

Szu Hui Ng is an Associate Professor in the Department of Industrial and Systems Engineering at the National University of Singapore. She holds BS, MS and PhD degrees in Industrial and Operations Engineering from the University of Michigan. Her research interests include quality and reliability modelling, and analysis, design of experiments, and simulation. She is a member of INFORMS and IEEE, and is a senior member of IIE.

References

Ababneh, I. (2008), ‘Availability-Based Noncontiguous Processor Allocation Policies for 2D Mesh-Connected Multicomputers’, *Journal of Systems and Software*, 81, 1081–1092.

Avizeinis, A. (1985), ‘The N-Version Approach to Fault-Tolerant Software’, *IEEE Transactions of Software Engineering*, SE-11, 1491–1501.

Berman, O., and Kumar, U.D. (1999), ‘Optimization Models for Recover Block Schemes’, *European Journal of Operation Research*, 115, 368–379.

Bhaskar, T., and Kumar, U.D. (2006), ‘A cost model for N-version programming with imperfect debugging’, *Journal of the Operation Research Society*, 7, 986–994.

Chatterjee, S., Misra, R.B., and Alam, S.S. (2004), ‘N-Version Programming With Imperfect Debugging’, *Computers and Electrical Engineering*, 30, 453–463.

Dai, Y.S., Xie, M., Poh, K.L., and Ng, S.H. (2004), ‘A Model for Correlated Failures in N-Version Programming’, *IIE Transactions*, 36, 1183–1192.

Dick, S., Bethel, C.L., and Kandel, A. (2007), ‘Software-Reliability Modeling: The Case for Deterministic Behavior’, *IEEE Transactions on Systems Man and Cybernetics Part A – Systems and Humans*, 37, 106–119.

Dugan, J.B., and Lyu, M.R. (1994), ‘System Reliability Analysis of an N-Version Programming Application’, *IEEE Transactions on Reliability*, 43, 513–519.

Gutjahr W.J. (2001), ‘A Reliability Model for Nonhomogeneous Redundant Software Versions With Correlated Failures’, *Computer Systems Science and Engineering*, 16, 361–270.

Gutjahr, W.J., and Uchida, G. (2002), ‘A branch-and-bound approach to the optimization of redundant software under failure correlation’, *Computers and Operations Research*, 29, 1773–1791.

Huang, C.Y., and Lo, J.H. (2006), ‘Optimal Resource Allocation for Cost and Reliability of Modular Software Systems in the Testing Phase’, *Journal of Systems and Software*, 79, 653–664.

Kapur, P.K., Bardhan, A.K., and Yadavalli, V.S.S. (2007), ‘On Allocation of Resources During Testing Phase of a Modular Software’, *International Journal of Systems Science*, 38, 493–499.

Kapur, P.K., Pham, H., Anand, S., and Yadov, K. (2011), ‘A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation’, *IEEE Transactions on Reliability*, 60, 331–340.

Knight, J.C., and Leveson, N.G. (1986), ‘An Experimental Evaluation of the Assumption of Independence in Multiversion Programming’, *IEEE Transactions on Software Engineering*, SE-12, 96–109.

Laprie, J.C., Arlat, J., Beounces, C., and Kanoun, K. (1990), ‘Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures’, *IEEE Computer*, 23, 39–51.

Littlewood, B., and Miller, D.R. (1989) ‘Conceptual Modeling of Coincident Failures in Multi-Version Software’, *IEEE Transactions on Software Engineering*, 15, 1596–1614.

Lee, C.H., Lee, S.M., and Park, D.H. (2005), ‘Evaluation of Software Availability for the Imperfect Software Debugging Model’, *International Journal of Systems Science*, 36, 671–678.

Levitin, G. (2005), ‘Optimal Structure of Fault-Tolerant Software Systems’, *Reliability Engineering and System Safety*, 89, 286–295.

Levitin, G., and Xie, M. (2006), ‘Performance Distribution of a Fault-Tolerant System in the Presence of Failure Correlation’, *IIE Transactions*, 38, 499–509.

Li, X., Xie, M., and Ng, S.H. (2012), ‘Multi-Objective Optimization Approaches to Software Release Time Determination’, *Asia-Pacific Journal of Operational Research*, 29, Article no. 1240019.

Lin, Y.K., Chang, P.C., and Fiodella, L. (2012), ‘Quantifying the Impact of Correlated Failures on Stochastic Flow Network Reliability’, *IEEE Transactions on Reliability*, 61, 692–701.

Lyu, M.R. (1996). *Handbook of Software Reliability Engineering*, New York: McGraw-Hill

Maciejewski, H., and Caban, D. (2008), ‘Estimation of Repairable System Availability Within Fixed Time Horizon’, *Reliability Engineering and System Safety*, 93, 100–106.

Mansour, H.F., and Dillon, T. (2011), ‘Dependability and Rollback Recovery for Composite Web Services’. *IEEE Transactions on Service Computing*, 4, 328–339.

Moranda, P.B. (1979), ‘Event-Altered Rate Models for General Reliability Analysis’, *IEEE Transactions on Reliability*, R-28, 376–381.

Nakagawa, Y., and Takenaka, I. (1991) ‘Error Complexity Model for Software Reliability Estimation’, *Transactions IEICE*, J74-D-I, 379–386.

Nicola, V.F., and Goyal, A. (1990), ‘Modeling of Correlated Failures and Community Error Recovery in Multiversion Software’, *IEEE Transactions on Software Engineering*, 16, 350–359.

Ouzineb, M., Nourelfath, M., and Gendreau, M., (2008), ‘Tabu Search for the Redundancy Allocation Problem of Homogenous Series–Parallel Multi-State Systems’, *Reliability Engineering and System Safety*, 93, 1257–1272.

Peng, R., Levitin, G., Xie, M., and Ng, S.H., (2011), ‘Optimal Defence of Single Object With Imperfect False Targets’ *Journal of the Operational Research Society*, 62, 134–141.

Pham, H. (1992), *Fault-Tolerant Software Systems: Techniques and Applications*. Washington, DC: IEEE Computer Society Press.

Pham, H. (2003). ‘Software Reliability and Cost Models: Perspectives, Comparison, and Practice’, *European Journal of Operation Research*, 149, 475–489

Pievatolo, A., Ruggeri, F., and Soyer, R. (2012), ‘A Bayesian Hidden Markov Model for Imperfect Debugging’,

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

Reliability Engineering and System Safety, 103, 11–21.

Randell, B. (1975), ‘System Structure for Software Fault Tolerance’, *IEEE Transactions on Software Engineering*, 1, 220–232.

Sergiy, A.V., Parnas, D.L., Mendiratta, V.B., and Murphy, E. (2008), ‘Computer Systems Availability Evaluation Using a Segregated Failures Model’, *Quality and Reliability Engineering International*, 24, 447–465.

Taboada, H.A., Espiritu, J.F., and Coit, D.W. (2008), ‘MOMS-GA: A Multi-Objective Multi-State Genetic Algorithm for System Reliability Optimization Design Problems’, *IEEE Transactions on Reliability*, 57, 182–191.

Tamai, T., and Torimitsu, Y. (1992), ‘Software Lifetime and Its Evolution Process Over Generations’, in *Proceedings of the International Conference on Software Maintenance*, pp. 63–69.

Teng, X., and Pham, H. (2002), ‘A Software Cost Model for Quantifying the Gain With Considerations of Random Field Environments’, *IEEE Transactions on Computers*, 53, 380–384

Tokuno, K. (2012). ‘Codesign-Oriented and User-Perceived Service Availability Measurement for Hardware/Software System’ *Asia-Pacific Journal of Operational Research*, 29, 1240024–1–1240024-223. doi: 10.1142(S02175959124 00246

Tokuno, K., and Yamada, S. (2003), ‘Relationship Between Software Availability Measurement and the Number of Restorations With Imperfect Debugging’, *Computers and Mathematics with Applications*, 46, 1115–1163

Tokuno, K., and Yamada, S. (2007), ‘User-Oriented and -Perceived Software Availability Measurement and Assessment With Environmental Factors’, *Journal of the Operation Research Society of Japan*, 50, 444–462

Wang, R.T. (2012), ‘A Dependent Model for Fault Tolerant Software Systems During Debugging’, *IEEE Transactions on Reliability*, 61, 504–515.

Wattanapongskorn, N., and Coit, D.W. (2007), ‘Fault-tolerant embedded systemdesign and optimization considering reliability estimation uncertainty’, *Reliability Engineering and System Safety*, 92, 395–407.

Wattanapongskorn, N., and Levitan, S.P. (2004), ‘Reliability Optimization Models for Embedded Systems With Multiple Applications’, *IEEE Transactions on Reliability*, 53, 406–416

Xie, M. (1991). *Software Reliability Modelling*, Singapore: World Scientific Publisher.

Xie, M., Dai, Y.S., Poh, K.L., and Lai, C.D. (2004), ‘Optimal Number of Hosts in a Distributed System Based on Cost Criteria’, *International Journal of Systems Science*, 35, 343–353.

Xie, M., Li, X., and Ng, S.H. (2011), ‘Risk-Based Software Release Policy Under Parameter Uncertainty’, *Proceedings of the Institution of Mechanical Engineers Part O – Journal of Risk and Reliability*, 225, 42–49.

Xie, M., and Yang, B. (2003), ‘A Study of the Effect of Imperfect Debugging on Software Development Cost’, *IEEE Transactions on Software Engineering*, 29, 471–473.

Xiong, C.J., Xie, M., and Ng, S.H. (2011), ‘Optimal Software Maintenance Policy Considering Unavailable Time’, *Journal of Software Maintenance and Evolution: Research and Practice*, 23, 21–33.

Yamachia, H., Tsujimuraa, Y., Kambayashia, Y., and Yamamoto, H. (2006), ‘Multi-Objective Genetic Algorithm for Solving N-Version Program Design Problem’, *Reliability Engineering and System Safety*, 91, 1083–1094

Zhang, Y.L. (2008), ‘A Geometrical Process Repair Model for a Repairable System With Delayed Repair’, *Computers and Mathematics with Applications*, 55, 1629–1643

Appendix

Let $F^i_{\alpha,\beta}(t)$ be the one-step transition probability that the process of version i at present time in state α will be in state β after elapsed time t. The transition probabilities can be written as:

$$F^i_{(j,0)(j,1)}(t) = 1 - e^{-\lambda_{i,j}t},\ j < N_i, \tag{A.1}$$

$$F^i_{(j,1)(j+1,0)}(t) = 1 - e^{-\mu_{i,j}t},\ j < N_i. \tag{A.2}$$

Suppose sub-version i is currently in working state and j faults have already been encountered and restored. Let $T^i_{(j,0)(k,0)}$ represent the first passage time of the sub-version i to the kth fault removal, where $j < k \le N_i$ and N_i is the number of initial faults of sub-version i. The distribution function of $T^i_{(j,0)(k,0)}$, denoted by $G^i_{(j,0)(k,0)}(t)$, can be expressed as:

$$G^i_{(j,0)(k,0)}(t) = F^i_{(j,0)(j,1)} {}^* F^i_{(j,1)(j+1,0)} {}^* G^i_{(j+1,0)(k,0)}(t), \tag{A.3}$$

for $j = 0, 1, 2, \ldots, k-1; k = 1, 2, 3, \ldots, N_i$, and $*$ symbolises the Stieltjes convolution. By taking the Laplace–Stieltjes (L-S) transform of both sides of (A.3), a new equation could be obtained:

$$\tilde{G}^i_{(j,0)(k,0)}(s) = \tilde{F}^i_{(j,0)(j,1)}(s)\tilde{F}^i_{(j,1)(j+1,0)}(s)\tilde{G}^i_{(j+1,0)(k,0)}(s), \tag{A.4}$$

where $\sim$ denotes the L-S transform. Using (A.1) and (A.2), one can further obtain:

$$\tilde{F}^i_{(j,0)(j,1)}(s) = \frac{\lambda_{i,j}}{s+\lambda_{i,j}}, \tag{A.5}$$

$$\tilde{F}^i_{(j,1)(j+1,0)}(s) = \frac{\mu_{i,j}}{s+\mu_{i,j}}. \tag{A.6}$$

Substituting (A.5) and (A.6) into (A.4), one can get

$$\tilde{G}^i_{(j,0)(k,0)}(s) = \frac{\lambda_{i,j}\mu_{i,j}}{(s+\lambda_{i,j})(s+\mu_{i,j})}\tilde{G}^i_{(j+1,0)(k,0)}(s). \tag{A.7}$$

An explicit solution for general j and k of (A.7) is hard to obtain. However, the time period that interests us is from the very beginning of software operation till its failure time. In other words, the special case with $j = 0$ is our research focus. According to the property of L-S transform, $\tilde{G}^i_{(k,0)(k,0)}(s) = 1$ for all $k \ge 0$. Starting from $j = 0$, by recursively replacing (A.7) with (A.5) and (A.6), (A.7) could be solved with boundary condition $\tilde{G}^i_{(k,0)(k,0)}(s) = 1$. The solution is

$$\begin{aligned}\tilde{G}^i_{(0,0)(k,0)}(s) &= \frac{\lambda_{i,0}\mu_{i,0}}{(s+\lambda_{i,0})(s+\mu_{i,0})}\tilde{G}^i_{(1,0)(k,0)}(s)\\ &= \frac{\lambda_{i,0}\mu_{i,0}\lambda_{i,1}\mu_{i,1}}{(s+\lambda_{i,0})(s+\mu_{i,0})(s+\lambda_{i,1})(s+\mu_{i,1})}\\ &\quad\times \tilde{G}^i_{(2,0)(k,0)}(s).\\ &= \cdots = \prod_{j=0}^{k-1}\frac{\lambda_{i,j}\mu_{i,j}}{(s+\lambda_{i,j})(s+\mu_{i,j})}\tilde{G}^i_{(k,0)(k,0)}(s)\\ &= \prod_{j=0}^{k-1}\frac{\lambda_{i,j}\mu_{i,j}}{(s+\lambda_{i,j})(s+\mu_{i,j})}\end{aligned} \tag{A.8}$$

For the probability $p^i_{(j,0)(k,0)}(t)$, it is very clear that $p^i_{(k,0)(k,0)}(t) = \exp\{-\lambda_{i,k}t\}$ and $p^i_{(N_i,0)(N_i,0)}(t) = 1$. Utilising the first passage time obtained above,

$$p^i_{(j,0)(k,0)}(t) = G^i_{(j,0)(k,0)}{}^* p^i_{(k,0)(k,0)}(t). \tag{A.9}$$

By taking the L-S transform of both sides of (A.9), one could obtain

$$\tilde{p}^i_{(j,0)(k,0)}(s) = \tilde{G}^i_{(j,0)(k,0)}(s)\tilde{p}^i_{(k,0)(k,0)}(s). \tag{A.10}$$

Similarly, the solution of a special case with $j = 0$ is the objective. Since $\tilde{p}^i_{(k,0)(k,0)}(s) = \frac{1}{s+\lambda_{i,k}}$ and $\tilde{p}^i_{(N_i,0)(N_i,0)}(s) = \frac{1}{s}$, by substituting (A.8) into (A.10), one can obtain

$$\begin{aligned}\tilde{p}^i_{(0,0)(k,0)}(s) &= \frac{1}{s+\lambda_{i,k}}\prod_{j=0}^{k-1}\frac{\lambda_{i,j}\mu_{i,j}}{(s+\lambda_{i,j})(s+\mu_{i,j})}, k < N_i,\\ \tilde{p}^i_{(0,0)(N_i,0)}(s) &= \frac{1}{s}\prod_{j=0}^{N_i-1}\frac{\lambda_{i,j}\mu_{i,j}}{(s+\lambda_{i,j})(s+\mu_{i,j})}.\end{aligned} \tag{A.11}$$

Equation (A.11) can be further formulated as

$$\tilde{p}^i_{(0,0)(k,0)}(s) = \prod_{j=0}^{k-1}(\lambda_{i,j}\mu_{i,j})\left\{\prod_{j=0}^{k-1}\frac{1}{(s-A_j)(s-B_j)}\right\} \times (s+\mu_{i,k}), k < N_i,$$

$$\tilde{p}^i_{(0,0)(N_i,0)}(s) = \prod_{j=0}^{N_i-1}(\lambda_{i,j}\mu_{i,j})\left\{\frac{1}{s}\prod_{j=0}^{N_i-1}\frac{1}{(s-A_j)(s-B_j)}\right\},$$

where $\left\{\begin{matrix}A_j\\B_j\end{matrix}\right.$ are two roots of the quadratic equation $x^2 + (\lambda_{i,j} + \mu_{i,j})x + \lambda_{i,j}\mu_{i,j} = 0$. By inversing the above equation, one can get

$$p^i_{(0,0)(k,0)}(t) = \left\{\prod_{l=0}^{k-1}\lambda_{i,l}\mu_{i,l}\right\}\sum_{j=0}^{k}\left\{\frac{A_j+\mu_{i,k}}{S_A(j,k)}\exp(A_jt) + \frac{B_j+\mu_{i,k}}{S_B(j,k)}\exp(B_jt)\right\}, k < N_i,$$

$$p^i_{(0,0)(N_i,0)}(t) = 1 + \left\{\prod_{l=0}^{N_i-1}\lambda_{i,l}\mu_{i,l}\right\}\sum_{j=0}^{N_i-1}\left\{\frac{\exp(A_jt)}{A_jS_A(j,1)} + \frac{\exp(B_jt)}{B_jS_B(j,1)}\right\}_i,$$

where

$$\left\{\begin{aligned}S_A(j,k) &= \left\{\prod_{l=0}^{k}(A_j - B_l\right\}\left\{\prod_{\substack{m=0\\m\neq j}}^{k}(A_j - A_m)\right\}\\ S_B(j,k) &= \left\{\prod_{\substack{l=0\\l\neq j}}^{k}(B_j - B_l)\right\}\left\{\prod_{m=0}^{k}(B_j - A_m\right\}\end{aligned}\right..$$

Downloaded by [NUS National University of Singapore] at 03:13 29 January 2015

View publication stats