A proposal of system to improve software reliability

Shinichi Funase (Faculty of Production Systems Engineering and Sciences, Komatsu University, shinichi.funase@komatsu-u.ac.jp)
Toshihiko Shimauchi (Faculty of Intercultural Communication, Komatsu University, toshihiko.shimauchi@komatsu-u.ac.jp)
Haruhiko Kimura (Faculty of Production Systems Engineering and Sciences, Komatsu University, haruhiko.kimura@komatsu-u.ac.jp)

Abstract
System failures include hardware failures and software failures. However, most of them are software failures. A redundant system is usually used to improve the reliability of hardware, but even if the same redundant principle is applied to a software system, each module (program) is a copy of the original, resulting in errors occurring in the same location without improving the reliability. Additionally, although a hardware failure can be detected relatively easily by human five senses, a software failure is difficult to be detected except in extreme cases such as when the system is stopped or gets out of control. To address these issues, this paper proposed a system that enables a robust redundancy for software. The proposed system monitors the control flow of the software and when an irregular flow is detected, the control flow shifts to another module with same function but different coding. The proposed redundant system is superior to existing single module system or other redundant systems in detecting errors and improving software reliability.

Key words
software reliability, MTTF, software redundancy, program error, software failure

1. Introduction

As the system becomes more complex and sophisticated, the impact of a failure on the system increases. Today, the one of the most serious challenges in system failure is software failures, against which various countermeasures have been implemented. However, automatic correction of "program error" is impossible and finding the cause of the error is extremely time consuming, which always bothers not only system engineers but also other people concerned with the software. In general, the decrease in software reliability correlates to two factors: the number of steps in the software and its difficulty. Moreover, as the scale of the program increases, detecting and correcting errors is rapidly increasing the difficulty. However, the current state of software inspection is still at the intermediate level, which relies on the intuition and experience of system engineers.

Also, few researches have been done to establish this field as a systematic discipline (Lyu, 1996; Operations Research Society of Japan, 2000; Jung et al., 2004; Spinellis, 2006; Yamada, 2011; Software Quality, 2020). In the programing theory, mathematical proof methods have been conducted to examine the correctness and halting problems. However, they did not solve the general problem of reliability of large-scale software systems (Floyd, 1967; Hetzel, 1973; Correctness, 2020). The goal in software reliability improvement is to minimize the number of bugs. However, reducing bugs is difficult and requires a huge amount of resources such as time and money. Existing studies utilized various methods other than bug reduction to improve software reliability. This paper utilizes redundant method in improving software reliability.

In this paper, software failures and reliability are theoretically examined, followed by a proposal of a system that improves the software reliability. This system uses methods developed by Funase et al. (2020; 2021) in detecting software failures. The system implements software redundancy using modules of the same function created by different programmers and different methods, which improves software reliability and mean time between failures (MTBF).

2. Software failure and reliability

2.1 Software failure

There is a serious discussion as to whether the word "failure" is appropriate for software problems. Some argues the word "error" is more appropriate than "failure". An "error" is a defect of a particular property or a static property, and a "failure" is a state in which a normal hardware or software suddenly reaches the end of its product life and becomes unusable. Unlike hardware, software has no lifespan. Therefore, the argument goes, "errors" are more appropriate.

However, there are counter arguments against this interpretation. If a thorough inspection is conducted, most of the bugs (logical errors) that frequently appeared prior to the inspection disappear, with occasional bugs appearing randomly. If something that was normal suddenly stops working, then it would be more appropriate to describe the situation as "the program has failed" rather than "there is an error in the program" (Hecht, 1975). The meaning of a general failure is "a system, a device, or a part loses its specified function". A hardware failure can be defined as damage to tangible components. However, in a software failure, there is no tangible damage. Therefore, in this paper, the time when the specified function is actually lost due to a functional defect or an algo-

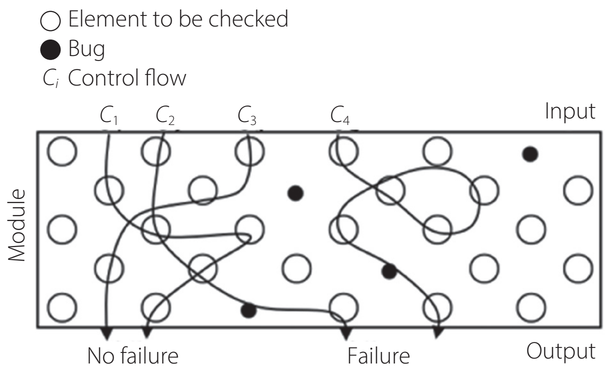


Figure 1: Diagram of Software failure

rithm error is judged as the time when the software failure occurs. In other words, when a program encounters a bug, a software failure is assumed to occur (Figure 1).

2.2 Software reliability

Software reliability is a probability that the software correctly achieves the intended function for the expected period under given operating conditions (Kimura and Oyabu, 2011).

Let $R(t)$ = reliability and $F(t)$ = unreliability, then

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

Let $f(t)$ = the derivative of $F(t)$ over time. Then,

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt}$$

$f(t)$ is a failure density function and indicates the increase in unreliability from time t per unit time Δt. This is the probability that the software operates normally from time 0 to t and fails within the time interval $[t, t + \Delta t]$. Therefore,

$$\int_0^\infty f(t)dt = 1$$

Let B = the number of bugs, C = the number of checks, and H = defect rate. Then H is calculated by

$$H = B / C$$

This equation shows the number of bugs per one check. The value can be considered as a failure probability. The failure rate $\lambda(t)$ is the failure rate per unit time Δt from the current t. By replacing the defect rate with the failure probability,

$$f(t) = R(t) \cdot \lambda(t)$$

and the following equation is obtained.

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{dR(t)}{dt} / R(T) \qquad (1)$$

When Equation (1) is integrated,

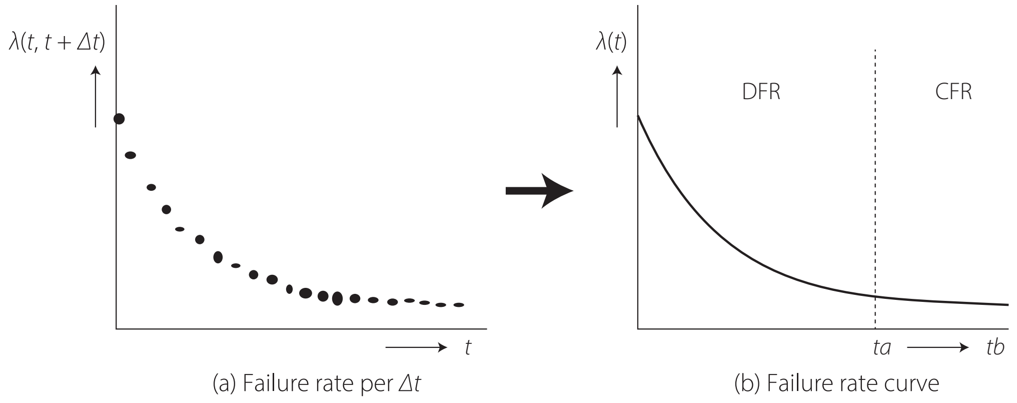
$$-\int_0^t \lambda(t)dt = \int_0^t \frac{1}{R(t)} \frac{dR(t)}{dt} dt = \int_0^t \frac{1}{R(t)} dR(t) = log_e R(t)$$

Therefore,

$$R(t) = e^{-\int_0^t \lambda(t)dt} \qquad (2)$$

The failure rate curve of $\lambda(t)$ is obtained by plotting the failure rate per unit time as shown in Figure 2. The curve consists of a decreasing failure rate (DFR) and a constant failure rate (CFR) curves. DFR is a type that is prone to failure in the initial stage, after which failure rarely occurs. When the software reaches this point, it faces with random errors whose probability is shown by CFR.

Specifically, in early DFR stage, bugs are frequently detected in routines used widely in the program. However, these bugs are easily detected and addressed in short span of time. In CRF stage, on the other hand, bugs are detected at routines less frequently used in a time tested program. These bugs significantly impact the overall error rates of the program and relatively difficult to detect and address. To reduce the occurrence of such errors and to address swiftly when



(a) Failure rate per Δt (b) Failure rate curve

Figure 2: Failure rate curve

one occurs, detailed attention is necessary in programming phase. In execution phase, an automatic inspection system is necessary for swift error detection and alternative route construction.

In Figure 2 (b), *tb* signifies the completion of the program as a marketable product. *tb* comes when certain amount of time passes after the error curve reaches CFR stage at t_a, where steady state has been accomplished. Until t_b is attained, debugging is continuously implemented. Now, let us consider the reliability of the program reaching this steady state.

Let $\lambda(t)$ in the steady state be λ.

$$\lambda\ (t \geqq t_a) = \lambda$$

Equation (2) can be modified as

$$R(t) = e^{-\int_0^t \lambda dt} = e^{-\lambda t}$$

Therefore, the mean time to failures (MTTF), which is the average time between a failure and the next failure, can be calculated as follows:

$$\begin{aligned}\mathrm{MTTF} &= \int_0^\infty tf(t)dt \\ &= \int_0^\infty t - \left(\frac{dR(t)}{dt}\right)dt \\ &= [-tR(t)]_0^\infty + \int_0^\infty R(t)dt \\ &= \int_0^\infty R(t)dt \\ &= \int_0^\infty e^{-\lambda t}dt \\ &= \left[-\frac{1}{\lambda}e^{-\lambda t}\right]_0^\infty \\ &= \frac{1}{\lambda}\end{aligned}$$

3. Proposed system

3.1 Idea

Now, we will introduce some ideas for improving software reliability. Generally, software redundancy is said to be meaningless. This is because when making modules E_1, E_2, ..., E_n, only E_1 is original and the rest are back up (copies) of this original module. In other words, if E_1 fails, all E_2, ..., E_n will fail.

But what if n-programmers create E_1, E_2, ..., E_n respectively? The modules of E_1, E_2, ..., E_n have the same function, but each has a different procedure and incorporates the individuality and skills of each programmer. This is the same as the difference in the reliability of the hardware device. In a case that only one programmer is available, this single programmer can create different modules with the same functionality by changing the method.

Based on these assumptions, this paper proposes the following system that applies the stand-by redundancy system. The difference from the general standby redundancy system is that the proposed system always starts from E_1. In a general standby redundant system, if E_1 fails, E_1 will never be used again. In the proposed system, if a failure is found in E_1, control shifts to the standby system until the process ends, after which the program starts from E_1 again. In detecting a failure in E_1, E_2, ..., E_n, the method proposed by Funase et al. (2020; 2021) is used. Accuracy of failure detection of E_1, E_2, ..., E_n is assumed to be 100 % and the reliability of the control shift switch to be 1.

3.2 MTFF of proposed system

Figure 3 shows the Shannon diagram of the redundant system proposed in this paper. In this system, if a failure is detected in module E_j, control shifts to E_{j+1}. For example, if a failure is found in E_1, control shifts to E_2. When the control shifts to the next module and the process completes, the program starts from E_1. Here, the state S_i indicates that the number of consecutive failures is i, and control is shifted to module E_{i+1}. λ_j is the failure rate of E_j, which is the transition probability from the state S_{j-1} to S_j per unit time Δ_t.

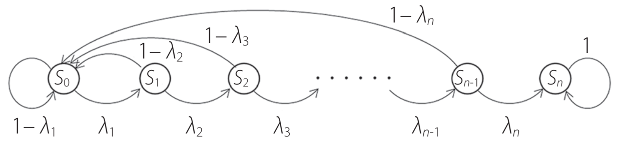


Figure 3: Shannon diagram of the proposed system

Now, discrete Markov processes is used to calculate the MTTF of the proposed system. In the transition probability matrix P between states, the transition probability from the state S_i to the state S_j is placed in n–i+1 rows and n–j+1 columns.

$$P = \begin{array}{c|cccccccc} & n & n-1 & n-2 & \cdots\cdots & 3 & 2 & 1 & 0 \\ n & 1 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ n-1 & \lambda_n & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 1-\lambda_n \\ n-2 & 0 & \lambda_{n-1} & 0 & \cdots\cdots & 0 & 0 & 0 & 1-\lambda_{n-1} \\ n-3 & 0 & 0 & \lambda_{n-2} & \cdots\cdots & 0 & 0 & 0 & 1-\lambda_{n-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 2 & 0 & 0 & 0 & \cdots\cdots & \lambda_3 & 0 & 0 & 1-\lambda_3 \\ 1 & 0 & 0 & 0 & \cdots\cdots & 0 & \lambda_2 & 0 & 1-\lambda_2 \\ 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & \lambda_1 & 1-\lambda_1 \end{array}$$

P can be disassembled into

$$P = \begin{bmatrix} I & O \\ Z & G \end{bmatrix}$$

whereas,

$$I = [1],\ O = [00 \cdots 0]$$

$$Z = \begin{bmatrix} \lambda_n \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 0 & \cdots\cdots & 0 & 0 & 1-\lambda_n \\ \lambda_{n-1} & 0 & \cdots\cdots & 0 & 0 & 1-\lambda_{n-1} \\ 0 & \lambda_{n-2} & \cdots\cdots & 0 & 0 & 1-\lambda_{n-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots\cdots & \lambda_2 & 0 & 1-\lambda_2 \\ 0 & 0 & \cdots\cdots & 0 & \lambda_1 & 1-\lambda_1 \end{bmatrix}$$

O is 1 row and *n* columns, *Z* is *n* rows and 1 column, and *G* is *n* rows and *n* columns.

After the state transition from S_{n-1} to S_n, the only thing left is the transition from the state S_n to S_n. This means that if the n^{th} module fails, the system will shut down completely. To consider only the state in which the system is operating, *G* is examined in the transition matrix.

The average number of times r_{ij}, which signifies that the state S_j appears when starting from the state S_i, is calculated by

$$r_{ij} = \sum_{t=0}^{\infty} P_{ij}(t)$$

$P_{ij}(t)$ is a probability of starting from the state S_i and transitioning to the state S_j after *t* units of time (Koyama, 1971). In this paper, the state is assumed to change in unit time and not to change until the first unit time is reached.

R aggregates all the possible combination of r_{ij} in single equation.

$$R = E + G + G^2 + \cdots$$

whereas

E: State transition probability matrix at the start (unit matrix of *n*-rows and *n*-columns)
G^t: State transition probability matrix after t unit time

R can be simplified as follows.

$$RG = G + G^2 + G^3 + \cdots$$
$$R - RG = E$$
$$R\,(E - G) = E$$
$$R = E\,(E - G)^{-1} = (E - G)^{-1}$$

Here, r_{ij} is an element of *R*'s *n-i* row and *n-j* column. The MTTF of the proposed system is the sum of $r_{00}, r_{01}, r_{02}, \cdots, r_{0n-1}$ multiplied by the unit time. That is, if the average number of times in which each state other than the state S_n appears when starting from the state S_0 is summed up, the average number of transitions in operation is obtained.

[Example 1] For example, consider the case of $n = 2$.

$$P = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_2 & 0 & 1-\lambda_2 \\ 0 & \lambda_1 & 1-\lambda_1 \end{bmatrix}, \; G = \begin{bmatrix} 0 & 1-\lambda_2 \\ \lambda_1 & 1-\lambda_1 \end{bmatrix}, \; E - G = \begin{bmatrix} 1 & \lambda_2 - 1 \\ -\lambda_1 & \lambda_1 \end{bmatrix}$$

$$(E - G)^{-1} = \frac{1}{\lambda_1 \lambda_2} \begin{bmatrix} \lambda_1 & 1-\lambda_2 \\ \lambda_1 & 1 \end{bmatrix}$$

Therefore,

$$\text{MTTF} = \frac{\lambda_1 + 1}{\lambda_1 \lambda_2}$$

3.3 Reliability of the proposed system

Although the discrete Markov process was used in the previous section, the proposed model is a continuous Markov process in which the state transition occurs not at discrete time points, but at random time points. It is a simple Markov process in which the probability of each state transition is always determined only from the current state regardless of the time required for the previous transition or modes of transition.

Let the system be in a state S_i at a certain time *t*, and let $a_{ij}dt$ be the probability of transition to another state S_j during the infinitesimal time *dt*.

When $j = i + 1$, then $a_{ij} = \lambda_j$
When $j = 0$, then $a_{ij} = 1 - \lambda_{i+1}$

If *dt* is very small, the probability that two or more transitions will occur in succession during this period is extremely small. The maximum number of state transition during this period is one.

Now, if the probability of being in a state S_i at time *t* is expressed by *Pi*(*t*),

$$P_i\,(t + dt) = P_i(t)\,(1 - \Sigma_{j\neq i}\, a_{ij}dt) + \Sigma_{j\neq i}\, P_i(t)\, a_{ij}dt \qquad (3)$$

Here,

$$\sum_{j\neq i} a_{ij}dt$$

indicates the probability of transition from state S_i to any other state within *dt* time.

Therefore,

$$1 - \sum_{j\neq i} a_{ij}dt$$

is the probability that no transition will occur.

Now, let us consider a model with $n = 2$. Figure 4 shows the transition probability per dt time between the states.

The following equation can be obtained from equation (3).

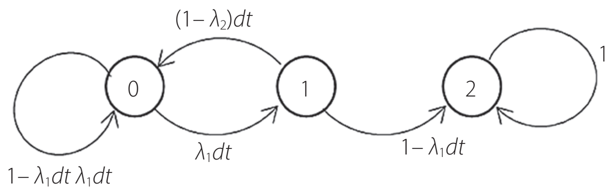


Figure 4: Shannon diagram showing the transition probability of each state per *dt* time

$$P_0(t+dt) = P_0(t)(1-\lambda_1 dt) + P_1(t)(1-\lambda_2)\,dt \tag{4}$$
$$P_1(t+dt) = P_1(t)\{1-(1-\lambda_2)dt - \lambda_2 dt\} + P_0(t)\,\lambda_1 dt \tag{5}$$
$$P_2(t+dt) = P_2(t)(1-0) + P_1(t)\,\lambda_2 dt \tag{6}$$

However, $P_0(0)=1$, $P_1(0)=0$, $P_2(0)=0$

From equation (4),

$$\begin{aligned} &P_0(t+dt) - P_0(t) = -P_0(t)\lambda_1 dt + P_1(t)(1-\lambda_2)\,dt \\ &P_0{}'(t) = -\lambda 1 P_0(t) + P_1(t)(1-\lambda_2) \end{aligned} \tag{7}$$

From equation (5),

$$\begin{aligned} &P_1(t+dt) - P_1(t) = -P_1(t)(1-\lambda_2)dt - P_1(t)\lambda_2 dt + P_0(t)\lambda_1 dt \\ &P_1{}'(t) = -P_1(t)\,(1-\lambda_2) - P_1(t)\lambda_2 + P_0(t)\lambda_1 \\ &= -P_1(t) + \lambda_2 P_1(t) - P_1(t)\lambda_2 + P_0(t)\lambda_1 \\ &= -P_1(t) + P_0(t)\lambda_1 \end{aligned} \tag{8}$$

From equation (6),

$$\begin{aligned} &P_2(t+dt) - P_2(t) = P_1(t)\lambda_2 dt \\ &P_2{}'(t) = P_1(t)\lambda_2 \end{aligned} \tag{9}$$

Using the Laplace transform, $P_2(t)$ can be obtained. The Laplace transform of $P_i(t)$ is expressed as follows:

$$L[P_i(t)] = g_i(S)$$

By applying the Laplace transform to the equations (7), (8), and (9),

$$\begin{cases} Sg_0(S) - P_0(0) = Sg_0(S) - 1 = -\lambda_1 g_0(S) + g_1(S)(1-\lambda_2) \\ Sg_1(S) - P_1(0) = Sg_1(S) = -g_1(S) + g_0(S)\lambda_1 \\ Sg_2(S) - P_2(0) = Sg_2(S) = g_1(S)\lambda_2 \end{cases}$$

Therefore,

$$g_2(S) = \frac{\lambda_1\lambda_2}{S\{S^2 + (\lambda_1 + 1)S + \lambda_1\lambda_2\}} = \frac{\lambda_1\lambda_2}{S(S-\alpha)(S-\beta)}$$

From this equation,

$$\alpha = \frac{1}{2}\,(-\lambda_1 - 1 + \sqrt{\lambda_1^2 + 2\lambda_1 + 1 - 4\lambda_1\lambda_2}\,)$$
$$\beta = \frac{1}{2}\,(-\lambda_1 - 1 - \sqrt{\lambda_1^2 + 2\lambda_1 + 1 - 4\lambda_1\lambda_2}\,)$$

are obtained. By applying Laplace inversion to $g_2(S)$,

$$\begin{aligned} P_2(t) &= \left[\frac{\lambda_1\lambda_2}{S(S-\alpha)}\,e^{St}\right]_{S=\beta} + \left[\frac{\lambda_1\lambda_2}{S(S-\beta)}\,e^{St}\right]_{S=\alpha} \\ &\quad + \left[\frac{\lambda_1\lambda_2}{(S-\alpha)\,(S-\beta)}\,e^{St}\right]_{S=0} \\ &= \frac{\lambda_1\lambda_2}{\alpha\beta(\alpha-\beta)}\left\{-\alpha e^{\beta t} + \beta e^{\alpha t} + \alpha - \beta\right\} \end{aligned}$$

is obtained and

$$R(t) = 1 - P_2(t) = \frac{\lambda_1\lambda_2}{\alpha\beta(\alpha-\beta)}\left\{-\alpha e^{\beta t} + \beta e^{\alpha t} + \alpha - \beta\right\} \tag{10}$$

If

$$A = \sqrt{\lambda_1^2 + 2\lambda_1 + 1 - 4\lambda_1\lambda_2}\,)$$
$$B = \ \ -\lambda_1 - 1$$

then, Equation (10) can be expressed as follows.

$$R(t) = \frac{1}{2A}\left\{(B+A)e^{\frac{1}{2}(B-A)t} - (B-A)e^{\frac{1}{2}(B+A)t}\right\}$$

Therefore,

$$\mathrm{MTTF} = \int_0^\infty R(t)dt = \frac{-4B}{B^2 - A^2} = \frac{\lambda_1 + 1}{\lambda_1\lambda_2}$$

This value matches with the MTTF obtained in the discrete Markov process. Table 1 shows a comparison of the reliability and MTTF between single module system and proposed dual module system ($n = 2$).

[Example 2] The single module system and the proposed dual module system are compared to calculate respective MTTF and reliability for working 10 consecutive days without errors. The failure rate of each module is assumed to be 0.1 per day.

From Table 1, the MTTFs of the single module and the dual module are respectively 10 days and 110 days. The reliabilities for working 10 consecutive days with the single module and dual modules are respectively 0.367879 and 0.920136.

[Example 3] Table 2 shows the MTTF of the proposed system with the number of modules n being from 2 to 9. The failure rate of each module is assumed to be 0.1 per day.

4. Conclusion

In this paper, a software failure is assumed to occur when a program encounters a bug. The paper proposed a system which detects a software failure based on the control flow and shifts the flow to another module with same function but programmed in different manner. The system with the redundancy improved the reliability of the program and MTTR. As for performance, as shown in Example 2, while the MTTF

Table 1: Comparison of reliability and MTTF between single module system and proposed dual module system

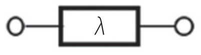
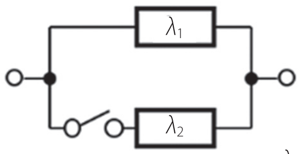
System configuration	Reliability	MTTF
Signal module  λ : Failure rate	$e^{-\lambda_t}$	$\frac{1}{\lambda}$
Proposed system  λ_i : Failure rate	$\frac{1}{2A}\left\{(B+A)e^{\frac{1}{2}(B-A)t} - (B-A)e^{\frac{1}{2}(B+A)t}\right\}$ whereas $A = \sqrt{\lambda_1^2 + 2\lambda_1 + 1 - 4\lambda_1\lambda_2}$ $B = \ -\lambda_1 - 1$	$\frac{\lambda_1 + 1}{\lambda_1\lambda_2}$

Table 2: MTTF of the proposed system

Number of modules (n)	MTTF (days)
2	110.0
3	1110.0
4	11109.8
5	111125.0
6	1112920.0
7	10652200.0
8	74565400.0
9	149131000.0

Note: The failure rate of each module is 0.1 per day.

for a single module was 10 days, that for the proposed system with 2 modules was 110 days. In addition, the reliability of a system that functioned normally for 10 days was 0.367879 for a single module, while that of the proposed system with 2 modules significantly improved to 0.920136. As the number of the module increased, MTTF improved by approximately 10 folds as shown in Example 3. This proportional increase will not continue since increase in the number of modules leads to the increase in similar function modules, which has almost no positive effect for robust redundancy.

Although hardware and software failures have different characteristics such as damage tangibility and failure detection difficulty, applying a hardware redundancy system to a software redundancy system has several implications. First, let us consider a parallel redundancy system. In this redundant system, control shifts to n devices at the same time, and if at least one device is in a normal state, control ends without any problem. In the case of software, the device is a module, but it is often difficult to determine whether a module is operating normally. Therefore, software redundancy in parallel redundancy is not possible.

Secondly, let us consider a stand-by redundancy system. In this redundant system, when the main operating device (first device) fails, a standby spare device acts as a substitute. The proposed system in this paper is similar to this standby redundant system with one significant difference: while the conventional standby redundant system never uses the failed device again, the proposed system reuses the module.

Finally, let us consider the m out of n redundancy system. This redundant system functions normally if the number of failed modules is m or less. A typical example is a 2 out of 4 redundant system used for a four-engined aircraft which can operate safely even if two engines fail. Even with this redundant system for a software, it is difficult to judge whether a module is functioning normally. This paper used a method developed by Funase et al. (Funase et al., 2020; 2021) which can detect a module malfunction with the accuracy of 80 % based on the control flow. The m out of n redundant system can also be used as a majority-decision redundant system. It is assumed that each module is created in a different way as in this paper. In other words, the result of the module with the same result or the frequency of the same state is adopted. However, many computers must be operated in parallel, and it takes a lot of time and effort to check each status.

From the above, it can be said that the software redundancy system proposed in this paper is superior to existing single module system or other redundant systems in detecting errors and improving software reliability.

References

Correctness (2020). Wikipedia. accessed on December 20, 2020. https://en.wikipedia.org/wiki/Correctness_(computer_science)

Funase, S., Simauchi, T., and Kimura, H. (2020). A proposal for run-time checks on command execution order of software program. *Studies in Science and Technology*, Vol. 9, No. 2, 149-152.

Funase, S., Simauchi, T., and Kimura, H. (2021). A theoretical analysis of automatic inspection of the control flow of computer programs. *Studies in Science and Technology*, Vol. 10, No. 1, to be published.

Floyd, R. W. (1967). Nondeterministic algorithms. *Journal of the ACM*, Vol. 14, No. 4, 636-644.

Hecht, H. (1975). Can software benefit from hardware experience? *Proceedings of Annual Reliability and Maintainability Symposium*.

Hetzel, W. C. (1973). *Program test methods*. Prentice-Hall.

Jung, H-W., Kim, S-G., and Chung, C-S. (2004). Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, Vol. 21, No. 5, 10-13.

Kimura, H. and Oyabu, T. (2011). *Introduction to information science*. Kyoritsu Shuppan. (in Japanese)

Koyama, A. (1971). *Markov processes and related fields*. Toyo Keizai. (in Japanese)

Lyu, M. R. (ed.) (1996). *Handbook of software reliability engineering*. McGraw Hill.

Operations Research Society of Japan (ed.) (2000). Software reliability. *Dictionary of Terminology*. JUSE Press. (in Japanese)

Software Quality (2020). Wikipedia. accessed on December 20, 2020. https://en.wikipedia.org/wiki/Software_quality.

Spinellis, D. (2006). *Code quality: The open source perspective*. Addison Wesley.

Yamada, S. (2011). *Introduction to software reliability*. Kyoritsu Shuppan. (in Japanese)

(Received: May 14, 2021; Accepted: June 5, 2021)