Availability and performability analysis for a service degradation process with condition-based preventive maintenance I – formulation and optimisation

Tadashi Dohi

Department of Information Engineering, Hiroshima University,
1-4-1 Kagamiyama, Higashi-Hiroshima 739-8527, Japan
E-mail: dohi@rel.hiroshima-u.ac.jp

Abstract: The preventive maintenance is very useful to improve effectively the service availability for software systems with service degradation. In this paper, we present a stochastic model to describe an operational software system, which consists of one operating system and multiple applications and provides a service in continuous time. Two kinds of rejuvenation strategies are taken, namely reconfiguration of applications as a corrective maintenance and reinstalation of an operating system as a preventive maintenance. We derive the optimal preventive rejuvenation schedules maximising the steady-state service availability and maximising the expected reward per unit time, by means of semi-Markov decision processes. Illustrative numerical examples are presented to give decision tables on the optimal software rejuvenation policies.

Keywords: software service; service degradation; software rejuvenation; condition-based maintenance; semi-Markov decision process; optimality; control-limit policy.


Biographical notes: Tadashi Dohi received his BSc (Engineering), MSc (Engineering) and PhD (Engineering) from Hiroshima University, Japan, in 1989, 1991 and 1995, respectively. In 1992, he joined the Department of Industrial and Systems Engineering, Hiroshima University, Japan as an Assistant Professor. Since 2002, he has been working as a Full Professor in the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. In 1992 and 2000, he was a Visiting Research Scholar in University of British Columbia, Canada and Duke University, USA, respectively, on leave of absence from Hiroshima University. His research areas include software reliability engineering, dependable computing and performance evaluation. He is a regular member of ORSJ, REAJ, IEICE, IPSJ and IEEE.

Copyright © 2014 Inderscience Enterprises Ltd.
1 Introduction

The value of service provided by computer systems is increasing day by day more than that of service equipments themselves. Hence, the notions of service reliability and service availability are becoming popular in our highly information-oriented society. As Tortorella (2005a, 2005b) mentioned, service reliability engineering enables service providers to design and operate a service delivery infrastructure in order to deliver services that meet stated service reliability requirement. Also, the above references defined the standard service reliability engineering procedure with the four steps;

1 setting service reliability requirements
2 sources of service reliability requirements
3 design for service reliability
4 service reliability monitoring.

In fact, since service reliability theory requires both analytical model-based and measurement-based frameworks to quantify the service failure mode, its concept can be involved completely in the usual dependable computing. More specifically, the software fault-tolerant computing is mainly classified into design diversity technique and environment diversity technique. The former is an engineering approach to realise the redundant design of equipments (hardware and software systems) for tolerating system failures, the latter achieves the temporal redundancy by changing the operation environment occasionally. Especially, if one focuses on the service provided by software, the environment diversity technique leads to an effective improvement to prevent a severe service failure (see Dohi, 2011). The typical examples are checkpointing to back up the service data, reconfiguration of application software and software rejuvenation.

As other examples on service reliability, Glossbrenner (1993) introduces the notions of reliability and availability for switched telecommunication services. Calabria et al. (1993) report a case study on the service dependability for real railway transit systems. Cristian (1993) introduces a distributed system service called availability management service which is responsible for ensuring that the critical services of a distributed system remain continuously available to users despite arbitrary numbers of concurrent node removals and node restarts caused by failures, maintenance and growth. Mainkar (1997) deals with transaction-processing systems based on user-perceived performance by introducing the so-called performability. Chu et al. (1998) consider a radio access telecommunications network and define what is known as service accessibility for a wireless access service. Reisinger (2004) discusses experimentally how the quality of service control by middleware can be established in a client-server system. Chan and Tortorella (2001) consider a simple probability model for sizing the spare
inventory to meet an end-to-end service availability objective. Dislis (2002) describes some of the challenges in providing continuous service and the impact of upgrade related outages for mobile telephone networks. Reinecke et al. (2004) treat a restart policy to speed up the completion of service as a representative application-level service dependability technique. Choi and Kim (2005) introduce the concept of performable availability as the probability that the systems are in one of some available states such that meet the minimum service requirement for cluster software systems. Dahlin et al. (2003) quantify the service availability for an end-to-end WAN system.

The main purpose of this paper is to develop a dynamic software rejuvenation model (see Adams, 1984; Castelli et al., 2001; Huang et al., 1995; Vaidyanathan and Trivedia, 2005) and derive the optimal software rejuvenation policy. In real time applications, actually, the condition-based preventive maintenance policies in static models on time (see Bao et al., 2005; Bobbio et al., 2001; Okamura et al., 2001, 2002, 2003a, 2003b, 2005; Wang et al., 2007) are difficult for use, because the on-line decision making to trigger the software rejuvenation is needed. On the other hand, the dynamic rejuvenation policy in Pfening et al. (1996) would be useful to take an action sequentially as observing the system state, although they never take account of an event of hard (system) failure. Apart from queueing models, Eto and Dohi (2005, 2006b) represent a service degradation process based on a right-skip free continuous-time Markov chain (CTMC) for a non-transaction-based software system. Along this direction, we also consider a service degradation model with preventive rejuvenation in Eto and Dohi (2006a), and give some mathematical results by means of semi-Markov decision process (see e.g., Chen and Trivedi, 2005). More specifically, we carry out not only service availability analysis but also service performability analysis of the service degradation process with preventive rejuvenation, and prove that the control-limit type of software rejuvenation policy is the best policy among all the stationary Markovian policies in our multistage service degradation model. Finally, illustrative numerical examples are presented to give decision tables on the optimal software rejuvenation policies.

2 Related work

The authors in Choi and Kim (2005) and Reinecke et al. (2004) focus on the typical software rejuvenation techniques (see Adams, 1984; Castelli et al., 2001; Huang et al., 1995; Vaidyanathan and Trivedia, 2005) for internet applications subject to system failures or unpredictable delays and cluster systems, respectively. It is common to consider that software faults are ideally removed during the debugging phase. Even if a piece of software has been thoroughly tested, however, it still may have some design faults that are yet to be revealed. Such software faults are called bohrbugs and may exist even in mature software such as commercial operating systems (see Gray, 1986; Grottke and Trivedi, 2007). Also, even mature software can be expected to have what are known as heisenbugs. These are bugs in the software that are revealed only during specific collusions of events. For example, a sequence of operations may leave the software in a state that results in an error on an operation executed next. Simply retrying a failed operation, or if the application process has crashed, restarting the process might resolve the problem. Another type of fault observed in software systems is due to the phenomenon of resource exhaustion (see Grottke and Trivedi, 2007). Operating system resources such as swap space and free memory available are progressively depleted
due to defects in software such as memory leaks and incomplete cleanup of resources after use. **Software aging** will affect the service performance of the application and eventually cause it to fail (see Adams, 1984; Castelli et al., 2001; Huang et al., 1995; Vaidyanathan and Trivedia, 2005). In other words, the software aging can be viewed as a degradation of a software system. A complementary approach to handle software aging and its related transient software failures, called **software rejuvenation**, is becoming popular (see Huang et al., 1995), and is regarded as a preventive maintenance policy that is particularly useful for counteracting the phenomenon of software aging, which is due to the phenomenon of resource exhaustion. In this paper it is assumed that the **service degradation** is caused by the software aging.

When software application executes continuously for long periods of time, some of the faults cause software to age due to the error conditions that accrue with time and/or load. Service degradation will also affect the performance of the application service and eventually cause it to fail. It is observed in widely-used communication software like Internet Explorer, Netscape and xrn as well as commercial operating systems and middleware. Huang et al. (1995) consider a stochastic model to trigger the software rejuvenation as a preventive solution. It involves stopping the running software occasionally, cleaning its internal state and restarting it. Cleaning the internal state of a software may involve garbage collection, flushing operating system kernel tables, reinitialising internal data structures, and hardware reboot. Since the seminal contribution by Huang et al. (1995), many stochastic models have been developed to trigger the software rejuvenation effectively to improve both of system availability and service availability (see Dohi et al., 2001, 2012), where the timing to trigger the software rejuvenation is given as a scheduled time. In other words, the software rejuvenation scheme assumed in the above literature is mainly based on the time-based rejuvenation policy. However, since the software aging can be characterised by some system resource parameters such as swap free, buffer size in physical RAM, etc., the time-based policy to trigger the software rejuvenation may not be appropriate in many cases.

Bobbio et al. (2001) and Okamura et al. (2003b) describe the cumulative damage models on the system resource and derive the condition-based optimal software rejuvenation policies. Xie et al. (2005) develop a two-level software rejuvenation scheme with service-level rejuvenation and box-level rejuvenation. Bao et al. (2005) and Wang et al. (2007) conduct workload-based analysis of software systems with rejuvenation under varying workload. Pfening et al. (1996) model a performance degradation process by the gradual decrease of the processing rate in a non-stationary Markovian queueing system, and formulate a determination problem of the optimal software rejuvenation schedule by a Markov decision process. Garg et al. (1998) consider a transaction-based software system, which involves arrival and queueing of jobs, and analyse both effects of aging; hard failures that result in an unavailability and soft failures that result in performance degradation. Okamura et al. (2001, 2002, 2003a, 2005) extend the queueing model by Garg et al. (1998) and analyse the transaction-based software systems with the control-limit type of rejuvenation software policies.

### 3 Service degradation model

Consider a software system which consists of one operating system and multiple applications. The system can provide a service in continuous time and, at the same
time, may deteriorate with time. Suppose that State 0 and State \( s + 1 \) are the normal (highly robust) state and the service down state, respectively. The service starts with State 0 at time 0 and makes a transition to \( s \) degradation levels \( k = 1', 2', \ldots, s' \) stochastically at any random time. At each change of state, the corrective maintenance will be made and the reconfiguration of some applications on the system will be made reactively by listing executing processes. Then, the service state becomes from \( k \) \( (= 1', 2', \ldots, s') \) to \( j \) \( (= 1, \ldots, s) \) after the corrective maintenance, where the constant set up time, \( k_j \), depending on the state \( j \) \( (= 0, 1, \ldots, s) \) and/or the corrective maintenance reward per unit time, \( c_1 \) \( (> 0) \), will be required. However, since the service can not become as good as new only by the reconfiguration of applications, the state just after the corrective maintenance makes a transition from \( j \) \( (= 1, \ldots, s - 1) \) to \( k \) \( (= 2', \ldots, s') \).

Figure 1  Markovian transition diagram of a service degradation model

Figure 2  Aggregated Markovian transition diagram

Suppose that the service level of software at time \( t \) is described by a semi-Markov process (SMP) with \( 2(s + 1) \) states and constant sojourn time \( k_j \) \( (j = 1, \ldots, s) \). Figure 1 illustrates the semi-Markovian transition diagram of a service degradation model. Since this SMP can be simplified to a right-skip free CTMC by aggregation of states \( k \) \( (= 1', 2', \ldots, s') \) and \( j \) \( (= 1, \ldots, s) \), it can be reduced to a simpler form depicted in Figure 2. More specifically, let \( \{N(t), t \geq 0\} \) be a right-skip free CTMC with space state \( I = \{0, 1, 2, \ldots, s, s + 1\} \), where the transition rate from \( i \) to \( j \) \( (i, j = 0, \ldots, s + 1, i < j) \) is given by \( \gamma_{i,j} \) \( (> 0) \) and \( \sum_{j=i+1}^{s+1} \gamma_{i,j} = \Gamma_i \) for all
\(i (= 0, 1, \ldots, s)\) in Figure 2. When the service failure occurs, the service is down and the state makes a transition from an arbitrary state \(j (= 0, 1, \ldots, s)\) to State \(s + 1\). Then, both the recovery operation and the reinstallation of operating system immediately start, where the time to complete the recovery operation is an independent and identically distributed (i.i.d.) random variable having the cumulative distribution function (c.d.f.) \(H_{s+1}(x)\) and mean \(1/\omega_{s+1} > 0\).

On the other hand, one makes a decision whether to trigger the software rejuvenation (reinstallation of operating system) as a preventive maintenance at the time instant when the state of software service changes from \(i (= 0, 1, \ldots, s)\) to \(j (= i + 1, i + 2, \ldots, s)\). If one decides to continue operation (Action 2), the state is monitored until the next change of state, otherwise, the software rejuvenation (Action 1) is preventively triggered, where the time to complete the rejuvenation is also an i.i.d. random variable with the c.d.f. \(H_i(x)\) and mean \(1/\omega_i > 0\), depending on the state \(i (= 0, 1, \ldots, s)\). When the service failure occurs, i.e., the service state becomes \(j = s + 1\), the recovery operation (Action 3) is taken. Let \(c_2 > 0\) and \(c_3 > 0\) be the rejuvenation reward per unit time and the recovery reward per unit time, respectively. In both periods of rejuvenation and recovery operation, the system operation is stopped. Also, it is assumed that the state-dependent reward \(r_i > 0\) is incurred per unit operation time for \(i = 0, 1, \ldots, s\). Let \(q_{i,j}(\delta)\) denote the probability that the service state changes from \(i\) to \(j\) under Action \(\delta (= 1, 2, 3)\). Then it is seen that

1. **Case 1 (rejuvenation):**
   \[
   q_{i,0}(1) = \int_0^\infty dH_i(t) = 1, \quad i = 0, 1, \ldots, s, 
   \]
   where the mean rejuvenation time (overhead) is given by
   \[
   h_i = \int_0^\infty tdH_i(t). 
   \]

2. **Case 2 (continuation of processing):**
   \[
   q_{i,j}(2) = \frac{\gamma_{i,j}}{\Gamma_1}, \quad i, j = 0, 1, \ldots, s + 1, i < j. 
   \]

3. **Case 3 (recovery from failure):**
   \[
   q_{i,0}(3) = \int_0^\infty dH_{s+1}(t) = 1, 
   \]
   where the mean recovery time (overhead) is given by
   \[
   h_{s+1} = \int_0^\infty tdH_{s+1}(t). 
   \]

After completing rejuvenation and recovery operations, the service state becomes as good as new, i.e., \(j = 0\) in equations (1) and (4), and the same cycle repeats again and again over an infinite time horizon. We define the time interval from the initial point to the completion of rejuvenation or recovery operation whichever occurs first, as one cycle.
4 Service availability analysis

4.1 Semi-Markov decision process

Observing the service state of software, we sequentially determine the optimal timing to trigger the software rejuvenation so as to minimise the steady-state service unavailability. Define the following functions:

- $\delta_A(i)$: action taken at state $i$, i.e.,
  \[ \delta_A(i) = \begin{cases} 
  1: & 0 \leq i \leq s \\
  2: & 0 \leq i \leq s \\
  3: & i = s + 1.
  \end{cases} \] (6)

- $G_A(i, \delta_A(i))$: expected service down time between successive two decision points when action $\delta_A(i)$ is taken at state $i$,
  \[ G_A(i, \delta_A(i)) = \begin{cases} 
  h_i: & \delta_A(i) = 1 \\
  k_i: & \delta_A(i) = 2 \\
  h_{s+1}: & \delta_A(i) = 3.
  \end{cases} \] (7)

- $\pi_A(i, \delta_A(i))$: total expected time between successive two decision points when action $\delta_A(i)$ is taken at state $i$,
  \[ \pi_A(i, \delta_A(i)) = \begin{cases} 
  h_i: & \delta_A(i) = 1 \\
  1/T_i: & \delta_A(i) = 2 \\
  h_{s+1}: & \delta_A(i) = 3.
  \end{cases} \] (8)

- $U(i)$: action space at state $i$.
- $v_A(i)$: value function at state $i$.
- $U_A^\infty$: service unavailability in the steady state, where $U_A^\infty$ denotes the minimum one.

From the preliminary above, the Bellman equation based on the principle of optimality (see Tijms, 1994) is given by

\begin{equation}
\begin{aligned}
  v_A(i) = \min_{\delta_A \in U(i)} \left[ G_A(i, \delta_A) - U_A^\infty \cdot \pi_A(i, \delta_A) \\
  + \sum_{j=0}^{s+1} q_{i,j}(\delta_A) \cdot v_A(j) \right].
\end{aligned}
\end{equation} (9)

It is well known that the software rejuvenation policy satisfying equation (9) is the best policy among all the Markovian policies (Tijms, 1994). To solve the above functional equation numerically, we can easily develop the well-known value iteration algorithm for the semi-Markov decision process. Define:
Availability and performability analysis

- $w_A(i, \delta(i))$ value function when action $\delta_A(i)$ is taken at state $i$
- $X_A(n) = \min_{i \in I} \{ v^n_A(i) - v^{n-1}_A(i) \}$
- $Y_A(n) = \max_{i \in I} \{ v^n_A(i) - v^{n-1}_A(i) \}$
- $\epsilon$ tolerance level
- $\tau$ design parameter satisfying $0 \leq \tau/h_r$ for all $i$, $\tau/\Gamma_i$ and $\tau/h_f \leq 1$ (see Tijms, 1994).

where $v^n_A(i)$ denotes the value function at $n^{th}$ iteration. Then, the value iteration algorithm is given in the following:

Value iteration algorithm

**Step 1:**

$$n := 0, \quad v_A^0(i) := 0. \quad (10)$$

**Step 2:**

$$w_{A}^{n+1}(i, 2) := \frac{k_i}{(1/\Gamma_i)} + \sum_{j=0}^{s+1} \tau_{\gamma_i,j} v_A^n(j) + \left( 1 - \sum_{j=0}^{s+1} \tau_{\gamma_i,j} \right) v_A^n(i),$$

$$w_{A}^{n+1}(i, 1) := \frac{h_i}{h_{s+1}} + \left( \frac{\tau}{h_{s+1}} \right) v_A^n(0) + \left( 1 - \frac{\tau}{h_{s+1}} \right) v_A^n(i),$$

$$v_{A}^{n+1}(i) := \min \{ w_{A}^{n+1}(i, 1), w_{A}^{n+1}(i, 2) \},$$

$$v_{A}^{n+1}(s+1) := \frac{h_{s+1}}{h_{s+1}} + \left( \frac{\tau}{h_{s+1}} \right) v_A^n(0) + \left( 1 - \frac{\tau}{h_{s+1}} \right) v_A^n(s+1).$$

**Step 3:** If $0 \leq Y_A(n) - X_A(n) \leq \epsilon X_A(n)$, then stop the procedure, otherwise, $n := n + 1$ and go to **Step 1**.

In general, it would be possible to derive the optimal software rejuvenation schedule by applying the above value iteration algorithm, if there exists a unique optimal solution. However, it is worth noting that an analytical approach to characterise the optimal rejuvenation policy, without solving the Bellman equation numerically, is possible by making some parametric (but reasonable) assumptions (see Eto and Dohi, 2005). In the following subsection, we investigate some mathematical properties for the optimal software rejuvenation policy and prove its optimality.

4.2 Optimality of control-limit policy

In a fashion similar to the previous result by the same authors (see Eto and Dohi, 2005), we prove the optimality of the control-limit type of policy. We make the following assumptions:

(A-1) $\Gamma_i$ is monotonically increasing in $i (= 0, 1, \cdots, s+1)$.

(A-2) For an arbitrary increasing function $f_j$: $\sum_{j=i+1}^{s+1} \gamma_{i,j} f_j/\Gamma_i$ is monotonically increasing in $i (= 0, 1, \cdots, j-1)$. 

For an arbitrary $x$, $\bar{H}_{s+1}(x) > \bar{H}_s(x)$, where in general $\bar{H}(\cdot) = 1 - H(\cdot)$.

$k_i$ is monotonically increasing in $i (= 0, 1, \cdots, s)$.

$k_i - h_i$ in monotonically increasing in $i (= 0, 1, \cdots, s)$.

The assumption (A-1) implies that the mean sojourn time in each state decreases, as the system deteriorates. The assumption (A-2) seems to be somewhat technical, but is intuitively reasonable. For instance, let $f_j$ be any cost parameter depending on state $j$. In this case, the expected down time incurred when the system state makes a transition, $\sum_{j=1}^{s+1} f_j(\gamma_{i,j}/\Gamma_i)$, tends to increase as the degraded level $i$ progresses. In the assumption (A-3), one expects in the sense of probability that the recovery time from system failure is strictly greater than the rejuvenation overhead. The assumption (A-4) means that the set up time increases, but according to (A-5) the motivation to trigger the software rejuvenation becomes stronger gradually, as the software system deteriorates more and more.

We give the main results of this paper.

**Lemma 3.1:** The function $v_A(i)$ is increasing in $i$.

**Proof:** It is evident from (A-3) to show that

$$
v_A(s, 1) = h_i + v(0) - UAh_i, \quad (11)
$$

$$
v_A(s + 1) = h_{s+1} + v(0) - UAh_{s+1}. \quad (12)
$$

Hence we have $v_A(s) \leq v_A(s + 1)$ immediately. Supposing that $v_A(i + 1) \leq v_A(i + 2) \leq \cdots \leq v_A(s) \leq v_A(s + 1)$ for an arbitrary $i$, from (A-2), it can be seen that

$$
\sum_j \frac{\gamma_{i,j}}{\Gamma_i} v_A(j) \leq \sum_j \frac{\gamma_{i+1,j}}{\Gamma_{i+1}} v_A(j). \quad (13)
$$

In Case 1 with $\delta_A(i+1) = 1$, we have

$$
v_A(i+1) - v_A(i) \geq (h_{i+1} - h_i) - UA(h_{i+1} - h_i) = 0. \quad (14)
$$

This implies that $v_A(i+1) \geq v_A(i)$. In Case 2 with $\delta_A(i+1) = 2$, we obtain

$$
v_A(i+1) - v_A(i) \geq k_{i+1} + \sum_j \frac{\gamma_{i+1,j}}{\Gamma_{i+1}} v_A(j) - \frac{UA}{\Gamma_{i+1}} - k_i

= \sum_j \frac{\gamma_{i,j}}{\Gamma_i} v_A(j) + \frac{UA}{\Gamma_i}

= (k_{i+1} - k_i) + \left( \sum_j \frac{\gamma_{i+1,j}}{\Gamma_{i+1}} v_A(j) - \sum_j \frac{\gamma_{i,j}}{\Gamma_i} v_A(j) \right) \quad (15)

+ \frac{UA}{\Gamma_i} - \frac{UA}{\Gamma_{i+1}}

\geq 0,
$$
Availability and performability analysis

which is due to (A-1), (A-2) and (A-4). Thus, it can be shown that \( v_A(i) \leq v_A(i+1) \). From the inductive argument, it can be proved that \( v_A(i) \leq v_A(i+1) \) for an arbitrary \( i \).

**Theorem 3.2:** There exists the optimal control limit \( N^* + 1 \) satisfying
\[
\delta_A(i) = \begin{cases} 
1 : \text{otherwise} \\
2 : i \leq N^*. 
\end{cases}
\]  
(16)

**Proof:**
\[
w_A(i,2) - w_A(i,1) = k_i + \sum_j \gamma_{i,j} v_A(j) - \frac{U_A}{\Gamma_i} - h_i - v_A(0) + UAh_i. 
\]  
(17)

From (A-1), (A-2), (A-5) and Lemma 4.1, it is seen that equation (21) is a monotonically increasing function of \( i \). Hence, the proof is completed.

From Theorem 3.2, the problem can be reduced to obtain the optimal control-limit \( N^* + 1 \) so as to minimise the steady-state service unavailability. Next, we formulate the steady-state service unavailability as a function of \( N \), i.e., \( UA_\infty = UA_\infty(N) \). Define
\[
R_{i,j} = \begin{cases} 
1 & i = j, \\
\sum_{k=i+1}^{j} \gamma_{i,k} R_{k,j}/\Gamma_i & \text{otherwise}, 
\end{cases}
\]  
(18)

\[
S_i = \begin{cases} 
\sum_{j=i+1}^{N} R_{i,j}/\Gamma_j & i \leq N, \\
0 & i > N, 
\end{cases}
\]  
(19)

where \( R_{i,j} \) is the transition probability from the state \( i \) to the state \( j \), and \( S_i \) denotes the mean time to trigger the software rejuvenation. Define the first passage time:
\[
T^* = \inf\{t \geq 0 : N(t) \geq N^* + 1\},
\]  
(20)

so that the random variable \( T^* \) is the first time when \( N(t) \) is greater than the level \( N^* + 1 \). Using the above notation, we can get the following result without the proof.

**Theorem 3.3:** The optimal software rejuvenation time is given by the first passage time \( T^* \), where the optimal threshold level \( N^* \) is the solution of \( \min_{0 \leq N < \infty} UA_\infty(N) \) and
\[
UA_\infty(N) = \sum_{j=1}^{N} R_{0,j} \left\{ k_j + \sum_{k=N+1}^{s} \frac{\gamma_{j,k} h_k}{\Gamma_j} + \frac{\gamma_{j,s+1} h_{s+1}}{\Gamma_j} \right\},
\]  
(21)

\[
S_0 + \sum_{j=1}^{N} \sum_{k=N+1}^{s} \frac{R_{0,j}}{\Gamma_j} \left\{ \gamma_{j,k} h_k + \gamma_{j,s+1} h_{s+1} \right\}.
\]
In equation (25), the function $U_{A\infty}(N)$ is formulated as the expected down time for one cycle divided by the mean time length of one cycle. The minimisation problem of $U_{A\infty}(N)$ with respect to $N (= 0, 1, 2, \cdots)$ is trivial. Define the difference of $U_{A\infty}(N)$ by $\phi(N) = U_{A\infty}(N + 1) - U_{A\infty}(N)$. If $\phi(N + 1) - \phi(N) > 0$, then the function $U_{A\infty}(N)$ is strictly convex in $N$. Further, if $\phi(N^* - 1) > 0$ and $\phi(N^*) \leq 0$, then there exist (at least one, at most two) optimal threshold level $N^*$ which minimises $U_{A\infty}(N)$, i.e., $U_{A\infty}^* = U_{A\infty}(N^*)$. In fact, the convex property of the function $U_{A\infty}(N)$ can be easily checked.

5 Service performability analysis

5.1 Semi-Markov decision process

Observing the service state of software system, we sequentially determine the optimal timing to trigger the software rejuvenation so as to maximise the steady-state service reward. Define the following functions:

- $\delta_R(i)$: action taken at state $i$, i.e.,
  \[
  \delta_R(i) = \begin{cases} 
  1 : 0 \leq i \leq s \\
  2 : 0 \leq i \leq s \\
  3 : i = s + 1.
  \end{cases}
  \]  (22)

- $G_R(i, \delta_R(i))$: expected service reward between successive two decision points when action $\delta_R(i)$ is taken at state $i$,
  \[
  G_R(i, \delta_R(i)) = \begin{cases} 
  -c_2 h_i : \delta_R(i) = 1 \\
  r_i(1 - k_i) - c_1 k_i : \delta_R(i) = 2 \\
  -c_3 h_{s+1} : \delta_R(i) = 3.
  \end{cases}
  \]  (23)

- $\pi_R(i, \delta_R(i))$: total expected time between successive two decision points when action $\delta_R(i)$ is taken at state $i$,
  \[
  \pi_R(i, \delta_R(i)) = \begin{cases} 
  h_i : \delta_R(i) = 1 \\
  1/T_i : \delta_R(i) = 2 \\
  h_{s+1} : \delta_R(i) = 3.
  \end{cases}
  \]  (24)

- $U(i)$: action space at state $i$,
- $v_R(i)$: value function at state $i \in I$,
- $U_{R\infty}$: expected service reward in the steady state, where $U_{R\infty}^*$ denotes the maximum one.

From the preliminary above, the Bellman equation based on the principle of optimality (Tijms, 1994) is given by

\[
v_R(i) = \max_{\delta_R \in U(i)} \left[ G_R(i, \delta_R) - U_{R\infty} \cdot \pi_R(i, \delta_R) + \sum_{j=0}^{s+1} q_{i,j}(\delta_R) \cdot v_R(j) \right].
\]  (25)
Similar to the service availability model, the software rejuvenation policy satisfying equation (25) is the best policy. To solve the above functional equation numerically, we can easily develop the well-known value iteration algorithm for the semi-Markov decision process. Define:

- $w_R(i, \delta_R(i))$: value function when action $\delta_R(i)$ is taken at state $i$
- $X_R(n) = \min_{i \in I} \{v_R^n(i) - v_R^{n-1}(i)\}$
- $Y_R(n) = \max_{i \in I} \{v_R^n(i) - v_R^{n-1}(i)\}$
- $\epsilon$: tolerance level
- $\tau$: design parameter satisfying $0 \leq \tau/h_r$ for all $i$, $\tau/\Gamma_i$ and $\tau/h_f \leq 1$ (see Tijms, 1994).

where $v_R^n(i)$ denotes the value function at $n^{th}$ iteration. Then, the value iteration algorithm is given in the following:

**Value iteration algorithm:**

**Step 1:**

$n := 0, \quad v_R^0(i) := 0.$

**Step 2:**

$w_R^{n+1}(i, 2) := \frac{r_i \left(\frac{1}{h_i} - k_i\right) - c_i k_i}{1/i} + \sum_{j=0}^{s+1} \tau \gamma_{ij} v_R^n(j) + \left(1 - \sum_{j=0}^{s+1} \tau \gamma_{ij}\right) v_R^n(i),$

$w_R^{n+1}(i, 1) := -\frac{c_i h_i}{h_{s+1}} + \left(\frac{\tau}{h_i}\right) v_R^n(0) + \left(1 - \frac{\tau}{h_i}\right) v_R^n(i),$

$R_{ij} := \begin{cases} 1 & i = j, \\ \sum_{k=i+1}^j \gamma_{i,k} R_{k,j}/\Gamma_i & \text{otherwise}, \end{cases}$

**Step 3:** If $0 \leq Y_R(n) - X_R(n) \leq \epsilon X_R(n)$, then stop the procedure, otherwise, $n := n + 1$ and go to **Step 2**.

Next, we give some mathematical properties for the optimal software rejuvenation policy, without the proofs for brevity.

**5.2 Structure of control-limit policy**

We formulate the service reward in the steady state as a function of $N$, i.e., $\text{UR}_\infty = \text{UR}_\infty(N)$. Define

$$R_{i,j} = \begin{cases} 1 & i = j, \\ \sum_{k=i+1}^j \gamma_{i,k} R_{k,j}/\Gamma_i & \text{otherwise}, \end{cases}$$

(27)
Theorem 4.1: There exists the optimal control limit \( N^* + 1 \) satisfying

\[
\delta R(i) = \begin{cases} 
1 & : \text{otherwise} \\
2 & : i \leq N^* 
\end{cases}
\]  

Theorem 4.2: The optimal software rejuvenation time is given by the first passage time \( T^* \), where the optimal threshold level \( N^* \) is the solution of \( \max_{0 \leq N < \infty} UR_{\infty}(N) \) and

\[
UR_{\infty}(N) = \frac{\sum_{j=1}^{N} R_{0,j} \left\{ r_j \left( \frac{1}{\Gamma_j} - k_j \right) - c_j k_j \right\}}{S_0 + \sum_{j=1}^{N} \sum_{k=N+1}^{s} \frac{R_{0,j} \left\{ \gamma_{j,k} h_k + \gamma_{j,s+1} h_{s+1} \right\}}{\Gamma_j}} - \sum_{j=1}^{N} R_{0,j} \left\{ \sum_{k=N+1}^{s} \frac{\gamma_{j,k} c_2 h_i + \gamma_{j,s+1} c_3 h_{s+1}}{\Gamma_j} \right\} 
\]

In equation (31), the function \( UR_{\infty}(N) \) is formulated as the expected service reward for one cycle divided by the mean time length of one cycle. The maximisation problem of \( UR_{\infty}(N) \) with respect to \( N \) (\( = 0, 1, 2, \ldots \)) is trivial. Define the difference of \( UR_{\infty}(N) \) by

\[
\phi(N) = UR_{\infty}(N + 1) - UR_{\infty}(N) \]

If \( \phi(N + 1) - \phi(N) > 0 \), then the function \( UR_{\infty}(N) \) is strictly convex in \( N \). Further, if \( \phi(N^* - 1) > 0 \) and \( \phi(N^*) \leq 0 \), then there exist (at least one, at most two) optimal threshold level \( N^* \) which maximises \( UR_{\infty}(N) \), i.e., \( UR_{\infty}^* = UR_{\infty}(N^*) \). In fact, the convex property of the function \( UR_{\infty}(N) \) can be checked numerically.

6 Illustrative examples

First of all, we numerically show the unique existence of the optimal software rejuvenation policy. Huang et al. (1995) suppose that the number of system states is only 3: normal operation, deterioration (failure probable state) and system down.
Availability and performability analysis

In this section we consider a generalised Markovian deterioration process with more degradation levels, where the model parameters used here are given by

\[ \gamma_0 = 0.05, \quad \gamma_1 = 0.04, \quad \gamma_2 = 0.03, \quad \gamma_3 = 0.02, \quad \gamma_4 = 0.01, \quad \gamma_5 = 0.006, \quad \gamma_6 = 0.009, \quad \gamma_7 = 0.008, \quad \gamma_8 = 0.006, \quad \gamma_9 = 0.005. \]

Figure 3 depicts the behaviour of the steady-state service unavailability with respect to \( N \). It can be shown that there is an optimal threshold level \( N^* = 2 \) in this example. If the state of software system in the CTMC, \( N(t) \), is described by a SMP with general transition probability, it is worth noting that Theorem 3.3 is still valid by replacing the transition rate \( \gamma_{i,j} \) by \( \gamma_{i,j}(t) \). In other words, if one can know all the transition probabilities completely, it is possible to obtain the analytical form of the service unavailability in the steady state and to minimise it with respect to the threshold level \( N \). Figure 4 depicts the behaviour of the expected service reward in the steady state with respect to \( N \), where \( r_0 = 4 \) (\( \$ \)), \( r_1 = 4 \) (\( \$ \)), \( r_2 = 3 \) (\( \$ \)), \( r_3 = 2 \) (\( \$ \)), \( r_4 = 1 \) (\( \$ \)), \( r_5 = 0.5 \) (\( \$ \)), \( c_1 = 1 \) (\( \$ \)), \( c_2 = 2.5 \) (\( \$ \)), \( c_3 = 5 \) (\( \$ \)). The other parameters are same as Figure 3. It can be shown that there is an optimal threshold level \( N^* = 1 \) in this example.

**Figure 3** Behaviour of the steady-state service unavailability with respect to \( N \)

**Figure 4** Behaviour of the service reward in the steady state with respect to \( N \)
Next we numerically derive the optimal software rejuvenation policy and its associated criterion. Figure 5 illustrates the Markovian (CTMC) transition diagram with absorption representing the deterioration process of the software service, where each transition rate is assigned on each arc. Suppose that the rejuvenation time and the recovery time from service failure are given by $\omega_1 = \omega_2 = \omega_3 = \omega_4 = \omega_5 = \omega_6 = 4.0 \text{ (hr}^{-1})$, and $\omega_7 = 2.0 \text{ (hr}^{-1})$, respectively. Also, it is assumed that $k_0 = 0.05 \text{ (hr)}$, $k_1 = 0.10 \text{ (hr)}$, $k_2 = 0.15 \text{ (hr)}$, $k_3 = 0.20 \text{ (hr)}$, $k_4 = 0.25 \text{ (hr)}$, $k_5 = 0.30 \text{ (hr)}$, $k_6 = 0.35 \text{ (hr)}$.

In Table 1, we obtain the so-called decision table to characterise the optimal software rejuvenation policy. From this result, it is optimal to trigger the software rejuvenation at the first time when the system state reaches to $i^* = N^* + 1 = 4$. Then the associated minimum service unavailability in the steady state is given by $UA_{\infty}^* = UA_{\infty}(3) = 0.0374026$.

Figure 5 An illustrative example with six degradation levels (CTMC)

Table 1 Decision table in CTMC case

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_4(i)$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Next, we consider more general case where the state transition is governed by a SMP. For better understanding the situation, we do not use the notation of transition rate $\gamma_{i,j}(t)$, but instead the following one:

- $exp(\lambda)$: exponential distribution with mean $1/\lambda$
- $Wei(\eta, m)$: Weibull distribution with scale parameter $\eta$ and shape parameter $m$.

Figure 6 depicts the SMP with respective transition probabilities, where $\omega = 4.0$ and $\chi = 2.0$ similar to the CTMC case. Table 2 presents the decision table for the SMP case, where the optimal threshold is given by $N^* = 3$ and the corresponding service unavailability in the steady state is $UA_{\infty}^* = 0.04328$. As mentioned before, the resulting rejuvenation schedule may not be strictly optimal. However, if we pay our attention to only the control-limit policy, this is of course optimal to maximises the service availability in the steady state.
Availability and performability analysis

Figure 6  An example with six degradation levels (SMP)

Table 2  Decision table in SMP case

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta_A (i) )</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

7  Conclusions

In this paper, we have considered a dynamic rejuvenation policy for a multistage degradation software system. We have formulated the underlying optimisation problem by a semi-Markov decision process and proved the optimality of control-limit type of software rejuvenation policy in the CTMC case. The result can be applied to the preventive maintenance problem with garbage collection for an application software, if the degradation level can be quantified by the total amount of memory leak and the temporal behaviour is modelled by a Markov or semi-Markov process.

References


