Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Software Technology 46 (2004) 989–1000
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Optimal and adaptive testing for software reliability assessment☆

Kai-Yuan Cai[a,*], Yong-Chao Li[a], Ke Liu[b]

[a]Department of Automatic Control, Beihang University (Beijing University of Aeronautics and Astronautics), Beijing 100083, China
[b]Institute of Applied Mathematics, Academy of Mathematics and System Sciences, Chinese Academy of Sciences, Beijing 100080, China

Available online 21 October 2004

Abstract

Optimal software testing is concerned with how to test software such that the underlying testing goal is achieved in an optimal manner. Our previous work shows that the optimal testing problem for software reliability growth can be treated as closed-loop or feedback control problem, where the software under test serves as a controlled object and the software testing strategy serves as the corresponding controller. More specifically, the software under test is modeled as controlled Markov chains (CMCs) and the control theory of Markov chains is used to synthesize the required optimal testing strategy. In this paper, we show that software reliability assessment can be treated as a feedback control problem and the CMC approach is also applicable to dealing with the optimal testing problem for software reliability assessment. In this problem, the code of the software under test is frozen and the software testing process is optimized in the sense that the variance of the software reliability estimator is minimized. An adaptive software testing strategy is proposed that uses the testing data collected on-line to estimate the required parameters and selects next test cases. Simulation results show that the proposed adaptive software testing strategy can really work in the sense that the resulting variance of the software reliability estimate is much smaller than that resulting from the random testing strategies. The work presented in this paper is a contribution to the new area of software cybernetics that explores the interplay between software and control.

Keywords: Software reliability assessment; Optimal testing; Adaptive testing; Controlled Markov chain; Adaptive control; Software cybernetics

1. Introduction

Software testing can be performed for reliability improvement by detecting and removing software defects or for reliability assessment or validation by freezing the code of the software under test. A testing strategy determines what or which test case should be selected and/or when software testing should be stopped. Optimal software testing is concerned with how to test software such that the underlying testing goal is achieved in an optimal manner. In other words, an optimal testing strategy should select the best test case each time during testing and/or stop software testing at the best time of instant. An example testing goal is to detect as many distinct defects as possible using only 100 tests. The corresponding optimal testing strategy depends on whether the detected defects are removed during testing or not. It also depends on the test cases selected already and the defects detected already to determine what test cases should be selected next. This is because the test cases selected already and the defects detected already give hints what test cases are most likely to detect a particular defect. Therefore, the feedback information of test data collected during software testing is essential for synthesizing the optimal testing strategy.

More testing goals can be considered. For example, how to detect and remove 20 defects (if any) using a minimal number of tests? In our previous work, we consider the optimal testing problem for reliability improvement in a closed-loop or feedback control framework [1–4]. Defects are removed upon being detected and thus software reliability is improved accordingly. The software under test serves as a controlled object and the software testing strategy severs as the corresponding controller.

☆ This paper is an extended and revised version of the following paper: K.Y. Cai, Y.C. Li, K. Liu, 'How to Test Software for Optimal Software Reliability Assessment', Proceedings the Third International Conference on Quality Software, IEEE Computer Society Press, 2003, pp. 32–39. Supported by the National Natural Science Foundation of China (60233020), the '863' Programme of China (2001AA113192) and the Aviation Science Foundation of China (01F51025).
* Corresponding author. Tel./fax: +86-10-8231-7328.
E-mail address: kycai@buaa.edu.cn (K.-Y. Cai).

0950-5849/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2004.07.006

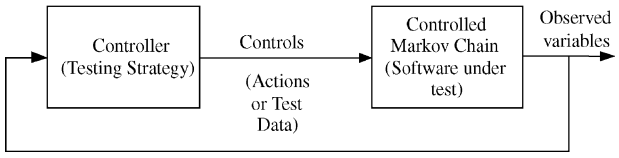

Fig. 1. Software testing as the optimal control problem.

The software under test and the software testing strategy make up a closed-loop feedback control system as shown in Fig. 1. The problem of identifying software defects at least cost or using a minimal number of tests becomes as an optimal control problem. Further, the software under test is modeled in the setting of controlled Markov chains (CMCs) and the theory of CMCs is used to solve the optimal control problem. This leads to the so-called CMC approach to software testing which was first proposed in 2000 [1]. The effectiveness of the CMC approach or the resulting adaptive testing (see Section 4) justifies the idea of software cybernetics, which is intended to explore the interplay between software and control [5–7].

In this paper, we discuss how to test software for reliability assessment. We assume that the code of the software under test is frozen or no detected defects are removed from the software under test during testing. Different testing strategies lead to different testing results, which in turn are used for reliability assessment for the software after testing. Optimal software reliability assessment means that the resulting reliability estimate is optimal or most trustable in some sense. We will show that the optimal software testing problem for reliability assessment can still be treated as the optimal control problem and dealt with in the setting of CMCs.

The rest of the paper is organized as follows. Section 2 formulates the software testing problem of concern. Section 3 presents the optimal software testing strategy. Section 4 proposes an adaptive strategy of software testing. Section 5 compares the proposed adaptive software testing strategy with the random testing strategies. Section 6 discusses some of the related problems of the proposed adaptive software testing strategy. Concluding remarks are given in Section 7.

2. Problem formulation

In the final phase of software testing, the software of concern is subjected to testing for validation or acceptance and reliability assessment is conducted by using the resulting test data. For highly reliable software, only few failures can be observed in the final phase of software testing, and the number of tests that can be applied is usually limited because of stringent schedule constraints. The software code is frozen. No debugging is invoked during testing even if software failures are observed. No modifications are applied to the software under test. The central problem of concern is how to apply a given number of tests such that the resulting software reliability estimate is best trustable. Since observed failures are few, point estimations of software reliability may suffer large fluctuations and it is desirable to keep the corresponding variance under control. Ideally, the variance of a software reliability estimator is minimized so that the resulting estimate may be as stable as possible.

In order to formulate the problem of concern, we have the following assumptions.

(1) The software under test or reliability assessment is frozen.
(2) The input domain of the software under test comprises m equivalence classes of test cases or input values, denoted $C_1, C_2, \ldots, C_m$.
(3) Each test case or input value will lead the software under test to failure or success, and

$$\Pr\{\text{a software failure is observed} | \text{a test case or input value of } C_i \text{ is applied}\} \equiv \theta_i \quad i = 1, 2, \ldots, m$$

(4) The output of the software under test when executed against a selected test case is independent of the history of testing.
(5) A software test includes selecting a test case or an input value from the input domain, executing the test case, collecting the resulting testing data, and updating the software reliability estimate if necessary. A total of n tests are allowed to be used to test the software.
(6) All actions or distinct software tests are admissible each time.
(7) The operational profile of the software under reliability assessment can be described as[1] $\langle C_i, p_i; i = 1, 2, \ldots, m \rangle$, that is, p_i denotes the probability that an input value is selected from C_i in the phase of software operation, and $\sum_{i=1}^{m} p_i = 1$.

Remarks

- The central problem here is how to select n tests one by one; each time a test or test case can be picked up from one of m equivalence classes.
- Assumption (5) means that a test comprises two major phases: a test case is selected, and the selected test case is executed.
- Assumption (4) means that upon being selected, the output of the software under test when executed against a selected test case is independent of the outputs of the software under test executed against all other test cases that were selected previously. A testing strategy determines which test case is selected at given time.

[1] This is the so-called Model I operational profile as described in Cai's book [8].

It has no effects on the output of the software under test when executed against the selected test case. However, this by no way means that selection of a test case is independent of testing history. Actually, testing history is used to guide the selection process of test cases, as will be seen in this section. We will revisit this simplifying assumption in Section 6.

- We can easily understand that different strategies of test case selection will lead to different testing results, making the variance of a software reliability estimator different. A reasonable testing goal is to use an unbiased estimator and minimize the corresponding variance. We should keep in mind that the true value of the reliability of the software under test cannot be known accurately until the software is put into operation for a sufficiently long time. What we can obtain in the testing phase is an estimate of the reliability of the software under test. What we can hope for is that the estimate lies in a confidence interval around the true value. Small variance implies that the resulting confidence interval is small. So, a desirable testing strategy should minimize the variance of a given software reliability estimator and thus make the resulting estimate as close as possible to the true value.

From the above assumptions we can see that the software reliability should be

$$R = \sum_{i=1}^{m} p_i(1-\theta_i) \tag{2.1}$$

or the software unreliability is

$$\rho = \sum_{i=1}^{m} p_i\theta_i \tag{2.2}$$

Suppose that among the n tests, n_i test cases are selected from C_i; that is, $\sum_{i=1}^{m} n_i = n$. Let

$$Z_{ij} = \begin{cases} 1 & \text{if a failure is observed when a test case is selected from } C_i \text{ for the } j\text{th time} \\ 0 & \text{otherwise} \end{cases}$$

Further, z_{ij} is realization of Z_{ij}. Based on the testing data, a natural estimator of θ_i is

$$\hat{\theta}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} z_{ij} \quad \text{or} \quad \frac{1}{n_i}\sum_{j=1}^{n_i} Z_{ij}$$

Accordingly, ρ can be estimated as

$$\hat{\rho} = \sum_{i=1}^{m} p_i\hat{\theta}_i = \sum_{i=1}^{m} \frac{p_i}{n_i}\sum_{j=1}^{n_i} Z_{ij} \tag{2.3}$$

Note that $\hat{\rho}$ is an unbiased estimator of ρ. Since $\{Z_{ij}\}$ is a series of independent random variables as a result of Assumption (4), the variance of the estimator $\hat{\rho}$ is

$$\mathrm{var}(\hat{\rho}) = \sum_{i=1}^{m}\left(\frac{p_i}{n_i}\right)^2 \sum_{j=1}^{n_i}\mathrm{var}(Z_{ij}) = \sum_{i=1}^{m}\left(\frac{p_i}{n_i}\right)^2 (n_i\theta_i(1-\theta_i))$$

Here, we use the fact that Z_{ij} follows a binominal distribution with parameter θ_i, that is, from Assumption (3) $Z_{ij}=1$ or a failure will be observed with probability θ_i when a test case of C_i is applied to the software under test. Note

$$\theta_i(1-\theta_i) = E\left[\frac{1}{n_i-1}\sum_{j=1}^{n_i}\left(Z_{ij} - \frac{1}{n_i}\sum_{k=1}^{n_i} Z_{ik}\right)^2\right]$$

where E denotes mathematical expectation. Therefore

$$\mathrm{var}(\hat{\rho}) = E\left[\sum_{i=1}^{m}\frac{p_i^2}{(n_i-1)n_i}\sum_{j=1}^{n_i}\left(Z_{ij} - \frac{1}{n_i}\sum_{k=1}^{n_i} Z_{ik}\right)^2\right] \tag{2.4}$$

Further note

$$\sum_{j=1}^{n_i}\left(Z_{ij} - \frac{1}{n_i}\sum_{k=1}^{n_i} Z_{ik}\right)^2 = \sum_{j=1}^{n_i}((Z_{ij})^2 - 2Z_{ij}L_i + (L_i)^2)$$

$$= \sum_{j=1}^{n_i}(Z_{ij} - 2Z_{ij}L_i + L_i^2) = n_iL_i(1-2L_i) + n_iL_i^2$$

$$= n_iL_i - n_iL_i^2$$

where

$$L_i = \frac{1}{n_i}\sum_{j=1}^{n_i} Z_{ij} \tag{2.5}$$

represents the number of failures observed in the n_i tests of equivalence class C_i. In this way we have

$$\mathrm{var}(\hat{\rho}) = E\left[\sum_{i=1}^{m}\frac{p_i^2L_i(1-L_i)}{n_i-1}\right] \tag{2.6}$$

That is, the variance of $\hat{\rho}$ is determined by $\{\langle p_i,n_i,L_i\rangle;\ i=1,2,\ldots,m\}$ and the ordering of tests applied is not involved. Since the software operational profile $\langle C_i,p_i;\ i=1,2,\ldots,m\rangle$ is predefined, we can reasonably use $\{\langle n_i,L_i\rangle;\ i=1,2,\ldots,m\}$ to represent the software state and ignore the ordering of tests applied. Since the true value of ρ is unknown, we have to trust an unbiased estimator of it that has the minimal variance. So, our goal is to derive a testing strategy that minimizes $\mathrm{var}(\hat{\rho})$. It is important to note that our goal is *not* to derive a testing strategy that minimizes $|\hat{\rho}-\rho|$.

Here, we should understand the role of variance for a software reliability estimator such as $\hat{\rho}$ given in Eq. (2.3). Suppose $\hat{\rho}_1$ and $\hat{\rho}_2$ are two unbiased estimators of ρ, with $\mathrm{var}(\hat{\rho}_1) < \mathrm{var}(\hat{\rho}_2)$. $\hat{\rho}_1$ implies that the true value of software reliability $\rho\in[\gamma_{11},\gamma_{12}]$, and $\hat{\rho}_2$ implies that

the true value of software reliability $\rho \in [\gamma_{21}, \gamma_{22}]$. However, $\mathrm{var}(\hat{\rho}_1) < \mathrm{var}(\hat{\rho}_2)$ implies that $|\gamma_{12} - \gamma_{11}| < |\gamma_{22} - \gamma_{21}|$. In other words, $\hat{\rho}_1$ is more accurate than $\hat{\rho}_2$ and is thus preferred. This is why we try to minimize the variance of a software reliability estimator, which is actually a function of software testing strategy.

$$q_{\xi_t \xi_{t+1}}(i) = \Pr\{\xi_{t+1} = (X_{t+1}, \eta_{t+1}^{(1)}, Y_{t+1}^{(1)}, \ldots, \eta_{t+1}^{(m)}, Y_{t+1}^{(m)}) | \xi_t = (X_t, \eta_t^{(1)}, Y_t^{(1)}, \ldots, \eta_t^{(m)}, Y_t^{(m)}), A_t = i\}$$

$$= \begin{cases} 1-\theta_i; & \text{if } X_t \neq 0,\ X_{t+1} = X_t - 1,\ \eta_{t+1}^{(i)} = \eta_t^{(i)} + 1,\ Y_{t+1}^{(i)} = Y_t^{(i)}, \text{ and } \eta_{t+1}^{(d)} \equiv \eta_t^{(d)},\ Y_{t+1}^{(d)} \equiv Y_t^{(d)} \quad \text{for } d \neq i \\ \theta_i; & \text{if } X_t \neq 0,\ X_{t+1} = X_t - 1,\ \eta_{t+1}^{(i)} = \eta_t^{(i)} + 1,\ Y_{t+1}^{(i)} = Y_t^{(i)} + 1, \text{ and } \eta_{t+1}^{(d)} \equiv \eta_t^{(d)},\ Y_{t+1}^{(d)} \equiv Y_t^{(d)} \quad \text{for } d \neq i \\ 1; & \text{if } X_t = 0,\ X_{t+1} = 0,\ \eta_{t+1}^{(i)} = \eta_t^{(i)},\ Y_{t+1}^{(i)} = Y_t^{(i)}, \text{ and } \eta_{t+1}^{(d)} \equiv \eta_t^{(d)},\ Y_{t+1}^{(d)} \equiv Y_t^{(d)} \quad \text{for } d \neq i \\ 0; & \text{otherwise} \end{cases} \tag{2.7}$$

We can also interpret the variance of a software reliability estimator from another view of point. Suppose both $\hat{\rho}_1$ and $\hat{\rho}_2$ are applied to 1000 distinct software systems (whose requirement specifications may or may not be the same) for reliability assessment, resulting in $\hat{\rho}_1^{(h)}$ and $\hat{\rho}_2^{(h)}$ for the hth software system, respectively. Then, $\mathrm{var}(\hat{\rho}_1) < \mathrm{var}(\hat{\rho}_2)$ implies that

$$\frac{1}{1000}\sum_{h=1}^{1000}\left(\hat{\rho}_1^{(h)} - \frac{1}{1000}\sum_{g=1}^{1000}\hat{\rho}_1^{(g)}\right)^2 < \frac{1}{1000} \times \sum_{h=1}^{1000}\left(\hat{\rho}_2^{(h)} - \frac{1}{1000}\sum_{g=1}^{1000}\hat{\rho}_2^{(g)}\right)^2.$$

In other words, $\hat{\rho}_1$ is more stable than $\hat{\rho}_2$.

Suppose the first test[2] is applied to the software under test at time $t=0$. The kth test is applied to the software under test at time $t=k-1$. Let $x_0 = n$ be the maximal number of tests that are allowed to be applied to the software under test, and X_t the remaining maximal number of tests that are allowed to be applied to the software under test at time t (inclusive). This means that $X_0 = x_0$, $X_1 = x_0 - 1, \ldots$. Let

$Y_t^{(i)}$ the number of failures revealed by tests of equivalence class C_i (action i) up to time $(t-1)$ (inclusive)
$\eta_t^{(i)}$ the number of tests of equivalence class C_i applied up to time $(t-1)$ (inclusive)
ξ_t $(X_t, \eta_t^{(1)}, Y_t^{(1)}, \ldots, \eta_t^{(m)}, Y_t^{(m)})$; $t = 0, 1, 2, \ldots$

In this way,[3] ξ_t can be treated as the software state at time t. In state ξ_t, there are X_t tests (including the one applied at time t) to be performed, and $\eta_t = \eta_t^{(1)} + \eta_t^{(2)} + \cdots + \eta_t^{(m)} = x_0 - X_t$ tests have been applied to the software under test. Among the η_t tests, $\eta_t^{(i)}$ are selected from equivalence class C_i, resulting in $Y_t^{(i)}$ failures.

Let

$A_t = i$ if a test is selected from equivalence class C_i at time t

Then from Assumptions (1)–(4) we have

$q_{\xi_t \xi_{t+1}}(i)$ implies that action i can make effects on $\eta_t^{(i)}$ and $Y_t^{(i)}$, but makes no effects on $\eta_t^{(d)}$ and $Y_t^{(d)}$ for $d \neq i$. The software testing must terminate if $X_t = 0$, or each of the states $(0, \eta_t^{(1)}, Y_t^{(1)}, \ldots, \eta_t^{(m)}, Y_t^{(m)})$ serve as an absorbing state.

3. Optimal software testing

Suppose the software under test is in state ξ_t. Note that ξ_t summarizes the test data and thus is a function of testing history. The problem now is which test case should be picked up so that $\mathrm{var}(\hat{\rho})$ given in Eq. (2.6) would be minimized. This is actually a kind of optimal testing problem mentioned in Section 1. Suppose the software defect detection rates $\theta_1, \theta_2, \ldots, \theta_m$ corresponding to the test cases of equivalence classes $1, 2, \ldots, m$, respectively, are known a priori.[4] Then, we can show in the following sections that there exists an optimal testing strategy that ensures that the variance of the given software reliability estimator is minimized. The resulting testing data are then used to derive an estimate of software reliability for the true value $R = 1 - \sum_{i=1}^{m} p_i \theta_i$.

In order to set the problem formulated in Section 2 into the context of CMCs,[5] we include a virtual action $\mathfrak{I}$ in the action set of the software under test. Then, the action set is $S = \{1, 2, \ldots, m, \mathfrak{I}\}$. Action $\mathfrak{I}$ is only admissible in the states with $X_t = 0$. It is the unique action that is admissible in these states, and leads the software to a virtual absorbing state. For the brevity of notation, the virtual absorbing state is still

[2] We confine ourselves to discrete-time domain. Note that in software reliability modeling, one needs to distinguish between continuous-time domain and discrete-time domain [9].

[3] In our previous works [1–3], the number of defects remaining in the software under test is used to define software state.

[4] Of course, if $\theta_1, \theta_2, \ldots, \theta_m$ are known a priori, then the true value of software reliability can be obtained directly and no testing is necessary for the purpose of software reliability assessment. Section 4 discusses how to test software if $\theta_1, \theta_2, \ldots, \theta_m$ must be estimated on-line. The optimal testing strategy can be used to evaluate how good a practical software testing strategy is.

[5] The preliminaries of CMCs can be found in Refs. [10,11]. Modeling of software testing processes in the setting of CMCs was first proposed by Cai in Ref. [1].

denoted as $\mathfrak{I}$, or, $\mathfrak{I}=(\mathfrak{I},\mathfrak{I},\mathfrak{I},\ldots,\mathfrak{I},\mathfrak{I})$. We have

$$q_{\xi_t\xi_{t+1}}(\mathfrak{I}) = \begin{cases} 1 & \text{if } X_t = 0, X_{t+1} = \eta_{t+1}^{(i)} = Y_{t+1}^{(i)} \equiv \mathfrak{I} \text{ for all } i \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Further, suppose an action a taken in state ξ incurs a cost of $W(\xi;a)$

$$W(X,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)};a) = \begin{cases} 0 & \text{if } a \in \{1,2,\ldots,m\} \\ \sum_{i=1}^{m} \dfrac{p_i^2 Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2} & \text{if } a = \mathfrak{I} \end{cases} \tag{3.2}$$

Then, the problem formulated in Section 2 can be treated as the first-passage problem of a CMC. Eqs. (2.7) and (3.1) define the probability transition law of the CMC, whereas Eq. (3.2) defines the corresponding cost structure. The CMC starts at state $(x_0,0,0,\ldots,0,0)$. Each time an action is taken from the action set $\{1,2,\ldots,m\}$, an incurring zero cost before the x_0 tests are used up. The x_0 tests end up in a proper state $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$. Then a final (virtual) action, $\mathfrak{I}$, is taken, incurring a cost of

$$\sum_{i=1}^{m} \frac{p_i^2 Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2}.$$

From the theory of CMCs we immediately conclude that there exists a deterministic stationary control policy (testing strategy) that minimizes $\mathrm{var}(\hat{\rho})$ (Eq. (2.6)) [10,11]. 'Deterministic' means that each time there exists a unique and optimal action (equivalence class of test cases) that should be selected. 'Stationary' means that the optimal action is only a function of software state and is independent of time. Here, we note that software state at time t is defined as $\xi_t = (X_t, \eta_t^{(1)}, Y_t^{(1)}, \ldots, \eta_t^{(m)}, Y_t^{(m)})$. Following the method of successive approximation in the theory of CMCs for obtaining the optimal testing strategy [10,11], we define

$$v_{\zeta+1}(\xi) = \min_{a:\ \text{admissible action in state } \xi} \left\{ W(\xi;a) + \sum_{\zeta \in \{S-\{\mathfrak{I}\}\}} q_{\xi\zeta}(a) v_\zeta(\zeta) \right\} \quad v(\xi) = \lim_{\zeta\to\infty} v_{\zeta+1}(\xi) \tag{3.3}$$

Please note that ζ does not denote the number of actions (test cases) selected. $v_{\zeta+1}(\xi)$ is defined according to the requirement of the method of successive approximation.

Proposition 1. *There holds*

$$\begin{aligned} v(x,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)}) = \min_{1\le i\le m} \{ & \theta_i v(x-1,\eta^{(1)},Y^{(1)},\ldots, \\ & \eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1,Y^{(i)}+1,\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}) \\ & + (1-\theta_i) v(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1, \\ & Y^{(i)},\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}) \} \end{aligned}$$

$$v(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)}) = \sum_{i=1}^{m} \frac{p_i^2 Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2}$$

Proof. Trivial from Eqs. (2.7) and (3.1)–(3.3). □

According to the theory of CMCs [10,11], Proposition 1 gives a clear picture of how to test software so that the variance of the given software reliability estimator is minimized. In state $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$, n tests have been performed already and software testing should be stopped. This corresponds to the observation that only $\mathfrak{I}$ or the virtual action can be taken. In state $(x,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ with $x\neq 0$, suppose action i is taken, then it will lead the software to state

$$(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1, Y^{(i)}+1,\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)})$$

with probability θ_i and state

$$(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)}, \eta^{(i)}+1,Y^{(i)},\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)})$$

with probability $1-\theta_i$. Among the m admissible actions, the action

$$\begin{aligned} \arg\min_{1\le i\le m} \{ & \theta_i v(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1, \\ & Y^{(i)}+1,\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}) \\ & + (1-\theta_i) v(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)}, \\ & Y^{(i-1)},\eta^{(i)}+1,Y^{(i)},\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}) \} \end{aligned}$$

should be taken. That is, in order to determine which action should be taken, the possible effects of all the admissible actions must be evaluated. In general, in order to determine the optimal action in state $(x,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$, we must take account of its possible following states

$$(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1, Y^{(i)}+1,\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}),$$

$$(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},Y^{(i-1)}, \eta^{(i)}+1,Y^{(i)},\eta^{(i+1)},Y^{(i+1)},\ldots,\eta^{(m)},Y^{(m)}),$$

$$(x-2, \eta^{(1)}, Y^{(1)}, \ldots, \eta^{(i-1)}, Y^{(i-1)}, \eta^{(i)}+1,$$
$$Y^{(i)}+1, \eta^{(i+1)}, Y^{(i+1)}, \ldots, \eta^{(j-1)}, Y^{(j-1)},$$
$$\eta^{(j)}+1, Y^{(j)}, \eta^{(j+1)}, Y^{(j+1)}, \ldots, \eta^{(m)}, Y^{(m)}),$$

$$(x-2, \eta^{(1)}, Y^{(1)}, \ldots, \eta^{(i-1)}, Y^{(i-1)}, \eta^{(i)}+1,$$
$$Y^{(i)}+1, \eta^{(i+1)}, Y^{(i+1)}, \ldots, \eta^{(j-1)}, Y^{(j-1)},$$
$$\eta^{(j)}+1, Y^{(j)}, \eta^{(j+-1)}, Y^{(j+-1)}, \ldots, \eta^{(m)}, Y^{(m)}),$$

and so forth.

Example 1. Suppose $A=\{1,2,3,4\}$, $x_0=100$, that is, the input domain of the software under test comprises four equivalence classes and 100 tests to be applied to the software. Further, $\theta_1=0.00025$, $\theta_2=0.0001$, $\theta_3=0.00035$, $\theta_4=0.0005$, and $p_1=0.1$, $p_2=0.3$, $p_3=0.5$, $p_4=0.1$. Suppose the current state of the software is (8,20,1,15,0,23,0,34,2). Then in order to determine the best action that should be taken in the current state, we need to obtain the value function v(8,20,1,15,0,23,0,34,2) by using Proposition 1. To this end, we must consider and compare the value functions of all its possible following states, including v(7,21,1,15,0,23,0,34,2), v(7,21,2,15,0,23,0,34,2), v(7,20,1,16,0,23,0,34,2), v(7,20,1,16,1,23,0,34,2), v(7,20,1,15,0,24,0,34,2), v(7,20,1,15,0,24,1,34,2), v(7,20,1,15,0,23,0,35,3), v(7,20,1,15,0,23,0,35,2), which each in turn needs to consider the value functions of its own following states. In this way, the value functions of all the possible states $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$, must be considered, which is evaluated as

$$\sum_{i=1}^{m} \frac{p_i^2 Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2}.$$

For the state (8,20,1,15,0,23,0,34,2) of our concern with $x=8$, the value functions are evaluated in Table 1 as a function of the action that is taken. Consequently, we have v(8,20,1,15,0,23,0,34,2)=min{2.972072, 3.094566, 3.164155, 2.974446}. This means that in the current state a test case should be selected from equivalence class 1.

Table 1
Value functions of example software state

Equivalence class	Value function of possible following state ($\times 10^{-5}$)	Value function of current state ($\times 10^{-5}$)
1	v(7,21,2,15,0,23,0,34,2)=4.149418 v(7,21,1,15,0,23,0,34,2)=2.969713	2.972072
2	v(7,20,1,16,1,23,0,34,2)=21.35323 v(7,20,1,16,1,23,0,34,2)=3.067137	3.094566
3	v(7,20,1,15,0,24,0,34,2)=30.78666 v(7,20,1,15,0,24,1,34,2)=3.067137	3.164155
4	v(7,20,1,15,0,23,0,35,3)=3.677539 v(7,20,1,15,0,23,0,35,2)=2.974094	2.974446

If the test leads the software to the state (7,21,1,15,0,23,0,34,2), then in order to make a right decision, we still need to consider the value functions of all its possible following states till $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$.

4. Adaptive software testing

In order to determine the best action that should be taken in a software state by using Proposition 1, the true values of $\theta_1,\theta_2,\ldots,\theta_m$ or software defect detection rates must be known a priori. However, in practice they are unknown and must be estimated by using the testing data collected on-line. This leads to an adaptive strategy of software testing, where the estimates of $\theta_1,\theta_2,\ldots,\theta_m$ are treated as true values and used for on-line decision-making.[6]

The fundamental idea of adaptive testing is that software testing is treated as an adaptive control problem [1,2,4]. At the beginning of software testing, the software tester has limited knowledge of the software under test and the capability of the test suite. As software testing proceeds, the understanding of the software under test and the test suite is improved. Accordingly, the testing strategy should also be improved. Fig. 2 shows a general diagram of adaptive testing. There are two feedback loops in the adaptive testing strategy. The first feedback loop occurs since the history of testing data is used to generate the next test cases by a given testing policy or test data adequacy criterion. The second feedback loop occurs since the history of testing data is also used to improve or change the underlying testing policy or test data adequacy criterion. The improvements may lead the random testing to switch from one test distribution (e.g. uniform distribution) to another test distribution (e.g. non-uniform distribution). It may also lead data flow testing to boundary value testing. In this paper the improvements lead to better test case selection scheme as a result of better estimates of $\theta_1,\theta_2,\ldots,\theta_m$.

However, from Proposition 1 and Example 1 we should understand that even if the underlying parameters are known, it is not easy to make exact decision-making based on $v(x,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$, and this is particularly true in the early phase of software testing where the number of remaining tests, x, is not small. Suppose $m=4$ or there are four distinct actions, then at the current state there are 100 remaining tests or $X_t=x=100$. In order to exactly determine which action is best in the current state, all the possible following states including various $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$, must be considered. This means that there are about 8^{100} states that must be considered for making a single exact decision. Obviously, computational complexity of this kind cannot be tolerant for on-line decision-making. To alleviate the computational

[6] This is just to follow the so-called certainty-equivalence principle in adaptive control [12].

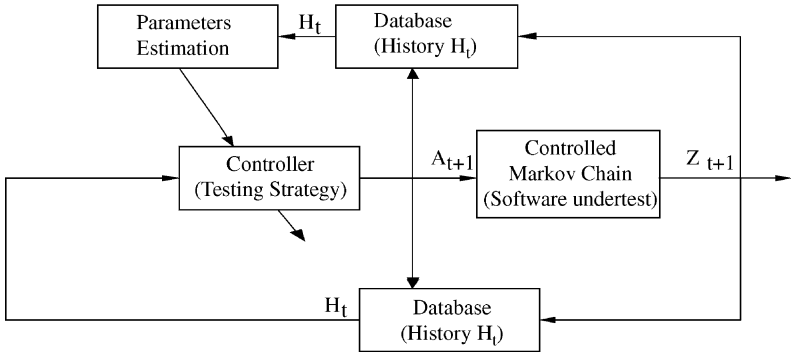

Fig. 2. Diagram of adaptive software testing.

complexity to a certain extent, we treat $(X_t-8, \eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ as $(0,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ and do approximate decision-making. That is, in order to evaluate the value function at the current state with $X_t \geq 8$, only the possible following states that are eight steps forward from the current state are considered.[7] The value functions in these following states are used to approximately evaluate the value function at the current state.

Further, from Proposition 1 we should also understand that there must be $\eta^{(i)} \geq 2$ for all $i=1,2,\ldots,m$. That is, each of the m admissible actions must be taken at least twice. Simply, we can select two test cases from each equivalence class at the beginning of testing. In summary, we have the following algorithm for adaptive software testing.

Step 0 Begin
Step 1 Let $t=0$, $X_0=x_0$, $\eta^{(1)}=\eta^{(2)}=\cdots=\eta^{(m)}=0$, $Y^{(1)}=Y^{(2)}=\cdots=Y^{(m)}=0$.
Step 2 Let $d=2$.
Step 3 Let $i=1$.
Step 4 Select a test case from equivalence class i and apply it to the software under test.
Step 5 Observe the output of the software under test when executed against the selected test case. If a failure is observed, then $Y^{(i)}=Y^{(i)}+1$.
Step 6 $\eta^{(i)}=\eta^{(i)}+1$.
Step 7 $t=t+1$.
Step 8 $X_t=X_{t-1}-1$.
Step 9 If $X_t=0$, go to Step 26.
Step 10 If $i=m$, go to Step 13.
Step 11 $i=i+1$.
Step 12 Go to Step 4.
Step 13 $d=d-1$.
Step 14 If $d>0$, go to Step 3.
Step 15 Let $\theta_i^{(t)}=Y^{(i)}/\eta^{(i)}$, $i=1,2\ldots,m$.
Step 16 Let $\theta_i=\theta_i^{(t)}$.
Step 17 Evaluate the value function $v(X_t,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ at the current state $(X_t,\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ by considering the various possible following states till $(\max\{0,X_t-8\},\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$. That is, $(\max\{0,X_t-8\},\eta^{(1)},Y^{(1)},\ldots,\eta^{(m)},Y^{(m)})$ is treated as an absorbing state whose value function is evaluated as

$$\sum_{i=1}^{m}\frac{p_i^2Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2}.$$

Step 18 Determine the action that should be taken at the current state as

$$\begin{aligned}a_t = \arg\min_{1\leq i\leq m}\{&\theta_i v(x-1,\eta^{(1)},Y^{(1)},\ldots,\eta^{(i-1)},\\&Y^{(i-1)},\eta^{(i)}+1,Y^{(i)}+1,\eta^{(i+1)},Y^{(i+1)},\ldots,\\&\eta^{(m)},Y^{(m)})+(1-\theta_i)v(x-1,\eta^{(1)},Y^{(1)},\ldots,\\&\eta^{(i-1)},Y^{(i-1)},\eta^{(i)}+1,Y^{(i)},\eta^{(i+1)},Y^{(i+1)},\ldots,\\&\eta^{(m)},Y^{(m)})\}.\end{aligned}$$

Step 19 Select a test case from equivalence class a_t and apply it to the software under test.
Step 20 Observe the output of the software under test when executed against the selected test case. If a failure is observed, then $Y^{(i)}=Y^{(i)}+1$.
Step 21 $\eta^{(i)}=\eta^{(i)}+1$.
Step 22 $t=t+1$.
Step 23 $X_t=X_{t-1}-1$.
Step 24 If $X_t=0$, go to Step 26.
Step 25 Go to Step 15.
Step 26 Stop testing.
Step 27 Evaluate $\rho=\sum_{i=1}^{m}p_i\theta_i$ and

$$\mathrm{var}(\rho)=\sum_{i=1}^{m}\frac{p_i^2Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2}.$$

Step 28 End

[7] Other approximate decision-making methods can also be adopted. An approximate decision-making method is acceptable if the resulting adaptive testing strategy outperforms the random testing strategy. What approximate decision-making methods are most acceptable is a topic deserving future investigation.

5. Comparisons with random testing

The adaptive testing strategy proposed in Section 4 is not an optimal testing strategy since the true values of parameters $\theta_1,\theta_2,\ldots,\theta_m$ are replaced by their estimates and only a part of all possible following states of a software state of concern is considered in making a decision. So the effectiveness of the adaptive software testing strategy must be justified. To this end, we do simulation and compare the adaptive testing strategy with the random testing strategies. In simulation we assume that the software under test is modeled by a binomial probability distribution that varies with equivalence class of selected test cases. If a test case of equivalence class i is selected, then a software failure is observed with probability θ_i and no software failure is observed with probability $1-\theta_i$. This probability distribution is used to determine either $Y^{(i)}=Y^{(i)}+1$ or $Y^{(i)}=Y^{(i)}$ each time.

Example 2. Let $x_0=3000$, $m=4$, $\theta_1=0.002$, $\theta_2=0.0015$, $\theta_3=0.0035$, $\theta_4=0.0005$, and $p_1=0.1$, $p_2=0.3$, $p_3=0.5$, $p_4=0.1$. In this way the true value of software reliability is $R=\sum_{i=1}^{4} p_i(1-\theta_i)=0.99755$. We do simulation for three different testing strategies. The first testing strategy is the adaptive testing strategy as defined in Section 4. The second testing strategy is the random testing strategy with uniform probability distribution being its test profile. In this strategy each time a test case is selected from equivalence class i ($i=$ 1,2,3, or 4) with probability 1/4 and no feedback of testing history is considered. The third testing strategy is the random testing strategy with operational profile $\{p_1,p_2,p_3,p_4\}$ being its test profile. In this strategy each time a test case is selected from equivalence class i ($i=$ 1,2,3, or 4) with probability p_i and no feedback of testing history is considered. For each testing strategy we do simulation for eight runs. In each run $x_0=3000$ tests are applied to the software under test (modeled by binomial probability distributions).

Table 2 tabulates the simulation results of the variance of R or ρ for the three testing strategies. From Table 2, we see that in the first run of simulation the adaptive testing strategy (testing strategy I) generates 5.570390×10^{-9} as the variance of software reliability estimate, and the mean value of the variance in the eight simulation runs is 6.542226×10^{-9}. Var(−) for the adaptive testing strategy is much smaller than that for the random testing strategy with operational profile being its test profile (testing strategy III), which in turn is smaller than that for the random testing strategy with uniform probability distribution being its test profile (testing strategy II). So, as far as the variance of software reliability estimates is concerned, the adaptive software testing strategy is preferred to the random testing strategies.

On the other hand, Table 3 tabulates the simulation results of software reliability estimates for the three testing strategies. The difference between the software reliability estimate and the true value of software reliability $R=$ 0.99755 for the random testing strategy with operational profile being its test profile is smaller than that for the random testing strategy with uniform probability distribution being its test profile, which in turn is smaller than that for the adaptive testing strategy. This is not desirable and we will revisit this in Section 6.

Example 3. In Example 2 the true values of $\theta_1=0.002$, $\theta_2=$ 0.0015, $\theta_3=0.0035$, $\theta_{14}=0.0005$ are relatively small in the sense of $x_0=3000$. Even if all the 3000 test cases are selected from equivalence class 1, there should be only six failures observed on average. In some run there may be no observed failure, leading the point estimate of θ_1 to zero. This may reduce the accuracy of the point estimate of R. In this example, we set $x_0=3000$, $m=4$, $\theta_1=0.012$, $\theta_2=0.015$, $\theta_3=0.035$, $\theta_4=0.005$, and $p_1=0.1$, $p_2=0.3$, $p_3=0.5$, $p_4=$ 0.1. In comparison with Example 2, the only difference is that the true values of θ_1, θ_2, θ_3, and θ_4 are each enlarged to 10 times. In this way the true value of software reliability is $R=\sum_{i=1}^{4} p_i(1-\theta_i)=0.9755$. As in Example 2, we do simulation for the three different testing strategies. Table 4 tabulates the simulation results of the variance of R or ρ for the three testing strategies, whereas Table 5 tabulates the corresponding simulation results of software reliability estimates. We can see that in terms of the variance of software reliability estimates, the adaptive testing strategy is better than the random testing strategy with the operational profile being its test profile, which in turn is better than the random testing strategy with the uniform probability distribution being its test profile. However, if the difference between the estimates of software reliability and the corresponding true value are considered, then the ordering relation should be reversed.

Table 2
Variance of software reliability estimates

Testing strategy	Var(1)	Var(2)	Var(3)	Var(4)	Var(5)	Var(6)	Var(7)	Var(8)	Var(−)	Unit
I	5.570390	10.01328	5.570390	6.682232	8.903679	4.457803	6.682232	4.457803	6.542226	$(\times10^{-9})$
II	2.041061	1.556177	2.841707	2.397153	1.029322	0.4375448	2.576874	0.8246939	1.71306625	$(\times10^{-6})$
III	4.230269	6.583483	4.429258	0.1217067	9.901007	0.1097958	6.592025	5.586617	7.55911375	$(\times10^{-7})$

Var(k): the variance in the kth simulation run, evaluated as

$$\sum_{i=1}^{4}\frac{p_i^2 Y^{(i)}(\eta^{(i)}-Y^{(i)})}{(\eta^{(i)}-1)(\eta^{(i)})^2};\quad \mathrm{Var}(-)=\sum_{k=1}^{8}\mathrm{Var}(k)/8.$$

Table 3
Software reliability estimates

Testing strategy	$R(1)$	$R(2)$	$R(3)$	$R(4)$	$R(5)$	$R(6)$	$R(7)$	$R(8)$	$R(-)$	Δ
I	0.99983290000000	0.99969930000000	0.99983290000000	0.99979950000000	0.99973270000000	0.99986640000000	0.99979950000000	0.99444440000000	0.99912595000000	0.15798205603729
II	0.99673600678678	0.99706109513222	0.99534468888454	0.99612994966449	0.99815821219748	0.99920885330822	0.99534884338360	0.99801376306630	0.99700017655295	0.05511738229161
III	0.99869791666667	0.99801213866584	0.99867034163930	0.99633483872923	0.99701170084656	0.99668213079175	0.99800910302879	0.99832842124917	0.99771832395216	0.01687373586887

$R(k)$: the software reliability estimate in the kth simulation run, evaluated as

$$\sum_{i=1}^{m} p_i\left(1-\frac{Y^{(i)}}{\eta^{(i)}}\right)_i; \quad R(-)=\sum_{k=1}^{8} R(k)/8; \quad \Delta=\left|\frac{R(-)-R}{R}\right|\times 100\%.$$

On the other hand, we note that in comparison with Example 2, for each of the three different strategies, Var(−) drops while Δ increases. Var(−) increases while Δ drops. This implies that there are inherent conflicting requirements between the mean value of software reliability estimates and the corresponding variance of software reliability estimates.

6. Discussion

One may be puzzled with the validity of the simulation results presented in Section 5 since the software reliability estimate for the adaptive software testing strategy deviates larger from the true value of software reliability than that for the random testing strategies. However, we can justify the validity of the simulation results in several perspectives. First, the estimator (2.3) is an unbiased estimator. It will eventually converge to the true value of the software unreliability as the number of tests increases. Different testing strategies will lead the variance of the resulting software reliability estimates to have different values, and it is natural to choose the testing strategy that makes the variance of the software reliability estimates minimal. Second, since the testing resource or the number of tests is limited, the resulting software reliability estimate must deviate from the true value of software reliability. How big the deviation is should be a function of software testing strategy. Different testing strategies may have different convergence rates of the software reliability estimate to the true value of software reliability. It is not surprising that while a testing strategy enjoys its advantage in terms of estimation variance, its convergence rate of estimate to true value may be comprised. Third, the adaptive testing strategy may be improved if more possible following software states are considered in the process of on-line decision-making, and this should lead to better estimates of software reliability.

An important assumption of the modeling approach to software testing process presented in this paper is that the output of the software under test when executed against a selected test case is statistically independent of the history of testing data. As we stress in Section 2, this should not be confused with the process of test case selections. The testing history is used to select next test cases. However, upon being selected, the output of executing the selected test case cannot be affected by the testing history. This assumption may be the main threat to the validity of the modeling approach to software testing processes. In many cases, the output of executing a selected test case is a function of testing history or a function of the current state of the software under test. However, the independence assumption of this kind is rather common in software reliability modeling for the sake of mathematical tractability [9,13,14]. For continuously running software there is a unique state called the START state at which execution begins for each test and that any history dependent state variables are

Table 4
Variance of software reliability estimates

Testing strategy	Var(1)	Var(2)	Var(3)	Var(4)	Var(5)	Var(6)	Var(7)	Var(8)	Var(−)	Unit
I	2.618484	0.06454310	0.06239769	0.07628956	0.06775562	3.033552	2.540128	3.947156	1.551288	$(\times 10^{-6})$
II	0.7521104	1.260517	1.310586	1.563847	1.117398	1.387172	1.171890	1.678649	1.280271	$(\times 10^{-5})$
III	7.945876	9.138955	6.419209	8.818513	5.786300	8.575424	8.397804	8.464784	7.943358	$(\times 10^{-6})$

Var(k): the variance in the kth simulation run, evaluated as

$$\sum_{i=1}^{4} \frac{p_i^2 Y^{(i)}(\eta^{(i)} - Y^{(i)})}{(\eta^{(i)} - 1)(\eta^{(i)})^2}; \quad \text{Var}(-) = \sum_{k=1}^{8} \text{Var}(k)/8.$$

cleared when the software arrive into the START state.[8] Of course we should note that the software state defined in this paper does not represent the values of physical variables used in real software systems. It can be interpreted as a test process state. The state adopted in the proposed CMC approach to software testing should widely be interpreted.

Another important assumption in this paper is that no detected defects are removed during software testing. This is not true in many critical embedded systems such as medical devices. In such cases no known failures of software can be tolerated unless they are of no critical concern to the user. Therefore, in the final as well as earlier phases of testing, any failure does lead to debugging. Thus, the real problem in such situations is "How does one estimate software reliability after causes for all known failures have been removed?" This problem is left to future investigation.

There is a huge amount of previous work on software reliability assessment [8,15,16], and Nelson [17] proposed a typical model for software reliability. Suppose the software operational profile is described as $\langle C_i, p_i; i = 1,2,\ldots,m\rangle$ and x test cases are generated according to the operational profile (that is, random testing strategy III described in Section 5 is applied). If y failures are observed out of the x tests, then according to the Nelson model, the software reliability is estimated as $1-(y/x)$. However, the Nelson model does not tell how good the software reliability estimate is. Normally, existing software reliability models use available testing data to assess software reliability. They do little about how to select test cases. Obviously, the modeling approach presented in this paper is dramatically different from existing approaches to software reliability modeling.

On the other hand, we note that there is research work that addresses the effects of testing on software reliability [18–20]. Software test processes have a strong effect on the delivered software reliability. However, the quantitative relationships between software test processes and the delivered software reliability are not clear. They have not been formalized yet. The modeling approach presented in this paper is a step towards formalizing software test processes for software reliability assessment. It treats software reliability assessment as a control problem by formalizing, quantifying and optimizing the underlying feedback mechanisms in software test processes.

The work presented in this paper exactly follows the general idea of software cybernetics that is intended to explore the interplay between software and control. In general, software cybernetics addresses issues and questions on [5–7]

(1) the formalization and quantification of feedback mechanisms in software processes and systems;
(2) the adaptation of control theory principles to software processes and systems;
(3) the application of the principles of software theories and engineering to control systems and processes; and
(4) the integration of the theories of software engineering and control engineering.

Among various related works on software cybernetics, the most related works should be those on software test process control by Cangussu et al. [21,22]. They developed a deterministic state variable model for feedback control/-management of the software test process. Partially related work is the so-called 'adaptive random testing' proposed by Chen et al. [23,24] to take account of the patterns of failure-causing inputs for guiding the test case selections.

A lot of work can be done for the modeling approach presented in this paper in the future. First, the relationship between the variance of the software reliability estimator and the corresponding estimate given by the adaptive testing strategy under limited testing resource should be studied. Second, the convergence behavior of software reliability estimate should be examined as the given number of tests increases. Third, efficient approximate algorithms should be developed to improve the adaptive testing strategy, whereas the computational complexity is under control. Fourth, case studies should be carried out to justify the proposed adaptive testing strategy. Fifth, besides the variance of software reliability estimator, other optimization criteria can be considered for the adaptive software testing strategy. A possible criterion is the one that combines the variance and the mean value of software reliability estimator together. Sixth, the deviation of the proposed adaptive testing strategy from the optimal testing strategy should be studied.

[8] An on-going case study with the Space program demonstrates that the simplifying assumption has minor impact on the reliability assessment. We will report this in a separate paper.

Table 5
Software reliability estimates

Testing strategy	$R(1)$	$R(2)$	$R(3)$	$R(4)$	$R(5)$	$R(6)$	$R(7)$	$R(8)$	$R(-)$	Δ
I	0.98379550000000	0.99802870000000	0.99809560000000	0.99766120000000	0.99792850000000	0.98112250000000	0.98429650000000	0.97943310000000	0.99004520000000	1.49105074320861
II	0.98662216979189	0.97743918348409	0.97570078635574	0.97137202254579	0.98013369994910	0.97660491882971	0.97828403020868	0.97081799308314	0.97712185053102	0.16625838349769
III	0.97574298837168	0.97221132475467	0.98033498280128	0.97284855691252	0.98212267570617	0.97358481055432	0.97380513873590	0.97388752472550	0.97556725032026	0.00689393339415

$R(k)$: the software reliability estimate in the kth simulation run, evaluated as

$$\sum_{i=1}^{m} p_i\left(1-\frac{Y^{(i)}}{\eta^{(i)}}\right)_i; \quad R(-)=\sum_{k=1}^{8} R(k)/8; \quad \Delta=\left|\frac{R(-)-R}{R}\right|\times 100\%.$$

Finally but not last, the applicability of the proposed adaptive testing strategy to systems other than software systems should be explored.

7. Concluding remarks

The CMC approach to software testing was originally proposed for optimal software reliability improvement. According to this approach, the software under test is modeled as a CMC and the theory of CMCs is used to synthesize the required software testing strategy that achieves the given testing goal (e.g. removing a given number of software defects at least cost) in an optimal manner. In this paper, we show that the CMC approach is also applicable to dealing with optimal software testing problem for software reliability assessment, where the code of the software under test is frozen and the number of tests is given. The software reliability assessment can be treated as a control problem too. First, we propose an unbiased software reliability estimator, derive its variance, and treat the variance as the optimization goal of software testing. The required software testing strategy should minimize the variance of the estimator in the sense of mathematical expectation. Second, we show that software states can be defined in terms of the numbers of applied tests and observed failures, and the ordering of applied tests can be ignored. Third, we show that the software under test for optimal software reliability assessment can be modeled as a CMC with a special cost structure. In this way, the required or optimal software testing strategy can be synthesized by using the theory of CMCs. Such an optimal software testing strategy uses the testing history to select next test cases on-line. Finally, since the parameters required by the optimal software testing strategy are unknown, we propose an adaptive software testing strategy in which the required parameters are estimated by using the testing data collected on-line. Simulation results show that the adaptive software testing strategy can really work in the sense that the resulting variance of the software reliability estimate is much smaller than those resulting from the random testing strategies.

The work presented in this paper is a step forward towards formalizing, quantifying and optimizing the relationships between software test processes and software reliability. It strengthens the applicability of the CMC approach to software testing and further justifies the idea of software cybernetics. It is a contribution to the new area of software cybernetics that explores the interplay between software and control.

References

[1] K.Y. Cai, A Controlled Markov Chains Approach to Software Testing, Working paper, 2000.
[2] K.Y. Cai, Optimal software testing and adaptive software testing in the context of software cybernetics, Information and Software Technology 44 (2002) 841–855.

[3] K.Y. Cai, Optimal stopping of multi-project software testing in the context of software cybernetics, Science in China (Series F) 46 (3) (2003) 335–354.
[4] K.Y. Cai, Y.C. Li, W.Y. Ning, Optimal software testing in the setting of controlled Markov chains, European Journal of Operational Research 162 (2) (2005) 262–289.
[5] K.Y. Cai, On the Concepts of Total Systems, Total Dependability and Software Cybernetics, (unpublished manuscript), Centre for Software Reliability, City University, London, Draft version, October 1994; revised version, July 1995
[6] K.Y. Cai, T.Y. Chen, T.H. Tse, Towards research on software cybernetics, Proceedings of Seventh IEEE International Symposium on High Assurance Systems Engineering, IEEE Computer Society Press, 2002. pp. 240–241.
[7] K.Y. Cai, J.W. Cangussu, R.A. De Carlo, A.P. Mathur. An overview of software cybernetics, Proceedings of the 11th International Workshop on Software Technology and Engineering Practice, 2003, IEEE Computer Society Press, 2004, pp. 77–86.
[8] K.Y. Cai, Software Defect And Operational Profile Modeling, Kluwer Academic Publishers, Boston, 1998.
[9] K.Y. Cai, Towards a conceptual framework of software run reliability modeling, Information Sciences 126 (2000) 137–163.
[10] C. Derman, Finite State Markovian Decision Processes, Academic Press, New York, 1970.
[11] O. Hernandez-Lerma, J.B. Lasserre, Discrete-Time Markov Control Processes: Basic Optimality Criteria, Springer, Berlin, 1996.
[12] O. Hernandez-Lerma, Adaptive Markov Control Processes, Springer, Berlin, 1989.
[13] A.L. Goel, Software reliability models: assumptions: limitations, and applicability, IEEE Transactions on Software Engineering SE-11 (12) (1985) 1411–1423.
[14] K.Y. Cai, C.Y. Wen, M.L. Zhang, A critical review on software reliability modeling, Reliability Engineering and System Safety 32 (1991) 357–371.
[15] M. Xie, Software Reliability Modeling, World Scientific, Singapore, 1991.
[16] Handbook of Software Reliability Engineering, in: M.R. Lyu (Ed.), McGraw-Hill, New York, 1996.
[17] E. Nelson, Estimating software reliability from test data, Microelectronics and Reliability 17 (1) (1978) 67–74.
[18] M.H. Chen, A.P. Mathur, V.J. Rego, Effect of testing techniques on software reliability estimates obtained using a time-domain model, IEEE Transactions on Reliability 44 (1) (1995) 97–103.
[19] Y.K. Malaiya, R. Karcich, The relationship between test coverage and reliability, Proceedings of Fifth International Symposium on Software Reliability Engineering (1994) pp. 186–195.
[20] P.G. Frankl, R.G. Hamlet, B. Littlewwod, L. Strigini, Evaluating testing methods by delivered reliability, IEEE Transactions on Software Engineering 24 (8) (1998) 586–601.
[21] J.W. Cangussu, R.A. DeCarlo, A.P. Mathur, A formal model for the software test process, IEEE Transactions on Software Engineering 28 (8) (2002) 782–796.
[22] J.W. Cangussu, R.A. DeCarlo, A.P. Mathur, Using sensitivity analysis to validate a state variable model of the software test process, IEEE Transactions on Software Engineering 29 (5) (2003) 430–443.
[23] T.Y. Chen, T.H. Tse, Y.T. Yu, Proportional sampling strategy: a compendium and some insights, Journal of Systems and Software 58 (2001) 65–81.
[24] T.Y. Chen, F.C. Kuo, R.G. Merkel, S.P. Ng, Mirror adaptive random testing, Proceedings of the Third International Conference on Quality Software, IEEE Computer Society Press, Silver Spring, MD, 2003. pp. 4–9.