On the Controlled Markov Chains Approach to Software Testing[1]

Kai-Yuan Cai
Department of Automatic Control
Beijing University of Aeronautics and Astronautics
Beijing 100191, China
kycai@buaa.edu.cn

Abstract - **The controlled Markov chains (CMC) approach to software testing is a relatively new approach that is distinctly different from conventional ones. It follows the idea of software cybernetics that explores the interplay between software and control, and treats software testing as a control problem. The software under test serves as a controlled object that is modeled as a controlled Markov chain, and the testing strategy as the corresponding controller. The software under test and the software testing strategy make up a closed-loop feedback control system. The CMC approach leads to an adaptive testing strategy if some parameters of the software under test are estimated on-line by using testing data to improve the corresponding software testing process. In this paper we revisit the CMC approach, present a new case study of adaptive testing, and discuss why the CMC approach or control-theoretic approach can work in practice. We further justify the effectiveness of the CMC approach by showing that the CMC approach can be applied to the optimal stopping problem of software testing as well.**

Keywords - **Software testing, controlled Markov chain, adaptive testing, adaptive control, optimal stopping problem, software cybernetics**

1 Introduction

Tremendous research efforts have been devoted to software test data adequacy criteria and software testing [17, 19-22, 29, 33-36, 39]. Many software testing techniques such as functional testing, boundary testing, path testing, state testing and mutation testing are available in practice [2, 4]. However software testing is still hard and arguably the least understood part of the software development process [18, 35]. An essential question has seldom be addressed: given a test goal (e.g., reliability improvement, reliability assessment), how to design an optimal testing strategy that achieves the goal.

The controlled Markov chains (CMC) approach to software testing is a relatively new approach that addresses the above question and is distinctly different from conventional ones. It follows the idea of software cybernetics [8, 9] and treats software testing as a control problem as shown in Figure 1 [7]. The software under test serves as a controlled object that is modeled as a controlled Markov chain, and the testing strategy as the corresponding controller. The software under test and the software testing strategy make up a closed-loop feedback control system. If some parameters of the software under test are estimated on-line to improve the software testing strategy, then the software testing strategy becomes as an adaptive one as shown in Figure 2 Adaptive testing is the counterpart of adaptive control in software testing [7]. It means that software testing strategy should be improved on-line by using the testing data collected during software testing as our understanding of the software under test is improved. There are two feedback loops in the process of adaptive testing. In the first (inner) feedback loop the software testing strategy uses the history of testing data to select or generate next test cases. In the second (outer) feedback loop the history of testing data is used to estimate the required parameters of the software under test and the testing strategy is updated or improved accordingly. A non-adaptive testing strategy specifies what test suite or what next test case should be selected, whereas an adaptive testing strategy specifies what next testing policy should be employed and thus in turn what test suite or next test case should be selected in accordance with the new testing policy.

1 Supported by the National Natural Science Foundation of China (Grant Number 60973006) and the Beijing Natural Science Foundation (Grant Number 4112033)

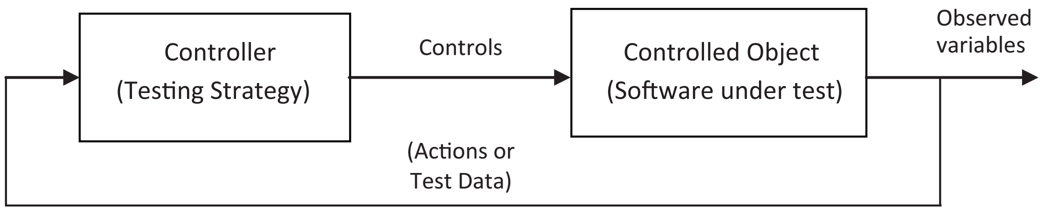

FIGURE 1
SOFTWARE TESTING AS A CONTROL PROBLEM

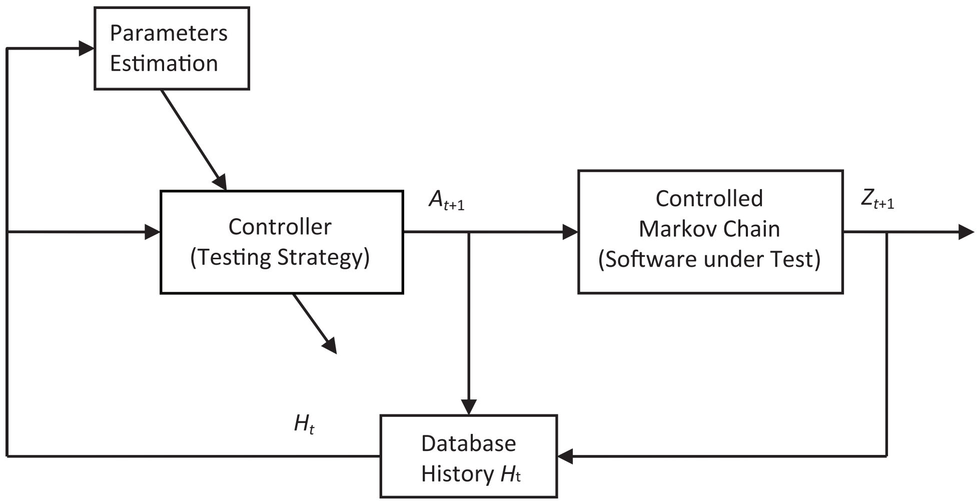

FIGURE 2 DIAGRAM OF ADAPTIVE TESTING

Here we note that adaptive testing discussed in this paper is applied for the purpose of software reliability improvement with detected software defects being removed from the software under test in the course of software testing [7]. However adaptive testing can also be applied for the purpose of software reliability assessment with software code being frozen with no defects being removed in the course of software testing [13].

In this paper we revisit the CMC approach, present a new case study of adaptive testing, and discuss why the CMC approach or control-theoretic approach can work in practice. We further justify the effectiveness of the CMC approach by showing that the CMC approach can be applied to the optimal stopping problem of software testing as well. Section 2 reviews the CMC approach and the adaptive testing strategy that is applied in our case study. Section 3 presents the case study of adaptive testing. Section 4 discusses related implications of experimental results. Section 5 applies the CMC approach to the optimal stopping problem of software testing. Concluding remarks are contained in Section 6. Five appendixes are included for related stuff and proofs that are mathematically involved.

2 THE CMC APPROACH AND ADAPTIVE TESTING

Let

$Y_t = j$;if the software under test contains j defects at time t,
$j = 0,1,2,\dots,N;\quad t = 0,1,2,\dots$

$$Z_t = \begin{cases} 1 & \text{if the action taken at time } t \text{ detects a defect} \\ 0 & \text{if the action taken at time } t \text{ doesn't detect a defect} \end{cases}$$

We have the following assumptions.

(1). The software contains $N \geq 1$ defects at the beginning ($t = 0$).

(2). An action taken at one time detects at most one defect.

(3). If a defect is detected, then it is removed immediately and no new defects are introduced; that is, $Y_t = j$ and $Z_t = 1$ mean $Y_{t+1} = j - 1$.

(4). If no defect is detected, then the number of remaining software defects remains unchanged; that is, $Y_t = j$ and $Z_t = 0$ mean $Y_{t+1} = j$.

(5). $Y_t = 0$ is an absorbing state; it is the target state.

(6). At every time there are always m admissible actions; the action set is $A = \{1, 2, \ldots, m\}$.

(7). Z_t depends only on the software state Y_t and the action A_t taken at time t;

$$\begin{aligned}
&\Pr\{Z_t = 1 | Y_t = j, A_t = i\} = j\theta_i \\
&\Pr\{Z_t = 0 | Y_t = j, A_t = i\} = 1 - j\theta_i \quad ; j = 1, 2, \ldots, N \\
&\Pr\{Z_t = 0 | Y_t = 0, A_t = i\} = 1 \qquad ; t = 0, 1, 2, \ldots
\end{aligned}$$

(8). Action A_t taken at time t incurs a cost of $W_{Y_t}(A_t)$, no matter whether or not it detects a defect;

$$W_{Y_t}(A_t = i) = \begin{cases} w_j(i) & \text{if } Y_t = j \neq 0 \\ 0 & \text{if } Y_t = 0 \end{cases}$$

(9). The cost of removing a detected defect is ignored.

Let τ be the first-passage time to state 0, and

$$W_{Y_t}(A_t = i) = \begin{cases} w_j(i) & \text{if } Y_t = j \neq 0 \\ 0 & \text{if } Y_t = 0 \end{cases} \tag{2.1}$$

where ω denotes a control policy (testing strategy). Our problem is to find the control policy (testing strategy) that minimizes $J_\omega(N)$. Such a policy removes all the N defects at the least expected cost.

The above assumptions and equation (1) define a simple controlled Markov chain as shown in Figure 2. Appendix 1 presents a brief introduction to controlled Markov chains. Intuitively, we may interpret θ_{A_t} as the software defect detection rate of action A_t. If action A_t is applied to the software in state j, $(j \neq 0)$, then the software remains in state j with probability $1 - j\theta_{A_t}$ and moves to state $j-1$ with probability $j\theta_{A_t}$. Upon entering into the target state 0, the software stays there forever. Let

$$v(1) = \min_{1 \le i \le m} \left\{ \frac{w_1(i)}{\theta_i} \right\} \tag{2}$$

$$v(j) = \min_{1 \le i \le m} \left\{ \frac{w_j(i)}{j\theta_i} + v(j-1) \right\} \qquad ; j = 2, 3, \ldots, N \tag{3}$$

We have shown in our previous work [7, 11] that the optimal action that should be taken is

$\arg_i v(1) = \arg_i \min_{1 \le i \le m} \left\{ \frac{w_1(i)}{\theta_i} \right\}$ in state $j = 1$ and is

$\arg_i v(j) = \arg_i \min_{1 \le i \le m} \left\{ \frac{w_j(i)}{j\theta_i} + v(j-1) \right\}$ in state $j \ge 2$.

That is, in state $Y_t \neq 0$, the optimal action is the one that minimizes $\frac{w_{Y_t}(i)}{\theta_i}$ with respect to i. In this way we obtain the CMC approach to software testing[2].

Equations (2) and (3) implicitly assume that the parameters of concern such as N and $\theta_1, \theta_2, \ldots, \theta_m$ are known. In practice they are unknown and need to be estimated in the course of software testing. What we can observe directly are the actions taken $\{A_t, t = 0, 1, 2, \ldots\}$ and the corresponding outputs $\{Z_t, t = 0, 1, 2, \ldots\}$. This leads to an adaptive testing strategy that is described as follows.

2 Two different approaches to software testing in case of $N = 1$, i.e., the conventional statistical approach and the CMC approach, are formulated in Appendixes 2 and 3, respectively.

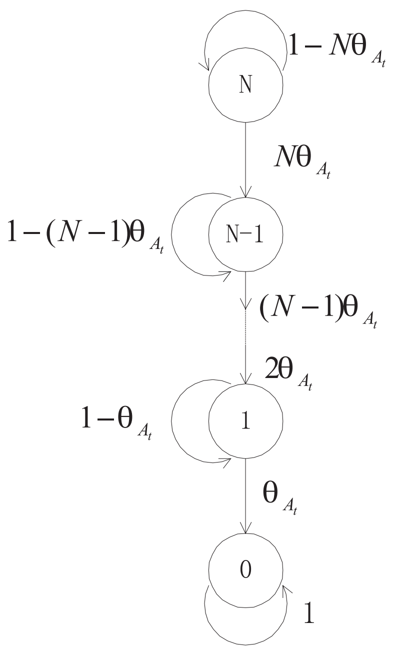

FIGURE 3
SOFTWARE STATE TRANSITION UNDER TEST

Denote $z_{-1}=0$. Let $\{z_t, t=0,1,2,\ldots\}$ be realization of $\{Z_t, t=0,1,2,\ldots\}$ and

$$r_t = \left(N - \sum_{j=-1}^{t-1} z_j\right)\theta_{a_t} \qquad (4)$$

where a_t is realization of A_t. In this way r_t represents the probability that action a_t detects a defect. We have

$$\Pr\{Z_t = z_t\} = (r_t)^{z_t}(1-r_t)^{1-z_t} \qquad ;t=0,1,2,\ldots \qquad (5)$$

Now given $z_0, z_1, \ldots, z_t$ and $a_0, a_1, \ldots, a_t$, the corresponding likelihood function is

$$\begin{aligned} L(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t) &= \Pr\{Z_0=z_0, Z_1=z_1,\ldots,Z_t=z_t\} \\ &= \prod_{j=0}^{t}\Pr\{Z_j = z_j\} \qquad (6) \\ &= \prod_{j=0}^{t}\left\{\left[\left(N-\sum_{k=-1}^{j-1} z_k\right)\theta_{a_j}\right]^{z_j}\left[1-\left(N-\sum_{k=-1}^{j-1} z_k\right)\theta_{a_j}\right]^{1-z_j}\right\} \end{aligned}$$

Suppose each of the m actions has been selected at least once up to time t. Then the $m+1$ parameters $N, \theta_1, \theta_2, \ldots, \theta_m$ can be estimated by maximizing the likelihood function $L(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t)$ (at least in theory). Denote the resulting estimates as

$N^{(t+1)}, \theta_1^{(t+1)}, \theta_2^{(t+2)}, \ldots, \theta_m^{(t+1)}$ .Or[3]

$$\begin{aligned} N^{(t+1)} &= N(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t) \\ \theta_i^{(t+1)} &= \theta_i(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t) \quad ;i=1,2,\ldots,m \end{aligned} \qquad (7)$$

By using the so-called certainty-equivalence principle or the method of substituting the estimates into optimal stationary controls [23, p38], we treat $N^{(t+1)}, \theta_1^{(t+1)}, \theta_2^{(t+2)}, \ldots, \theta_m^{(t+1)}$ as the true values of the corresponding parameters at time $t+1$ and take the optimal action based on these values. Consequently, we obtain the following adaptive control policy (adaptive testing strategy).

Step 1 Initialize parameters. Set[4]

$$z_{-1}=0,\ N^{(0)}=N_0,\ \theta_i^{(0)}=\theta_{0i} \quad ;i=1,2,\ldots,m$$

$$Y_0 = N^{(0)}, \quad A_0 = \arg\min_{1\le k\le m}\left\{\frac{w_{Y_0}(k)}{\theta_k^{(0)}}\right\}, \quad t=0$$

Step 2 Estimate parameters by maximizing[5]

$$L(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t)$$

(equation (2.6)) and obtain

$$\begin{aligned} N^{(t+1)} &= N(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t) \\ \theta_i^{(t+1)} &= \theta_i(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t) \quad ;i=1,2,\ldots,m \end{aligned}$$

3 A commonly used way of parameter estimation is to differentiate with respect to each parameter of concern and obtain a system of nonlinear equations. However sometimes the system of nonlinear equations may not have a solution. So, a more general way of doing parameter estimation is to maximize $L(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t)$ directly.

4 $\arg\min_x f(x) = x_0$ means $\min_x f(x) = f(x_0)$

5 If θ_i doesn't appear in $L(z_0,z_1,\ldots,z_t;a_0,a_1,\ldots,a_t)$, or action i has not been taken in the previous actions up to time t, then we let $\theta_i^{(t+1)} = \theta_i^{(t)}$.

Step 3 Decide the optimal action

$$A_{t+1} = \arg\min_{1 \le k \le m} \left\{ \frac{w_{Y_{t+1}}(k)}{\theta_k^{(t+1)}} \right\}$$

Step 4 Observe the testing result $Z_{t+1} = z_{t+1}$ activated by the action A_{t+1}.

Step 5 Update the current software state

$$Y_{t+1} = N^{(t+1)} - \sum_{j=-1}^{t} z_j$$

Step 6 Set $t = t + 1$.

Step 7 If a given testing stopping criterion is satisfied, then stop testing[6]; otherwise go to Step 2.

Here we note that in the above adaptive testing strategy, all available testing data are employed to do parameter estimation. However this is not essential. Adaptive testing can also use fixed-memory testing data to do parameter estimation and achieve desirable testing performance [11].

3 CASE STUDY

In order to better understand the effectiveness of the adaptive testing strategy presented in Section 2, we present a case study in this section.

3.1 Subject Program

In this case study, both the adaptive testing strategy and the random testing strategy were applied to test a real subject program that was written in Visual Basic language and comprised over 1,000 lines of code. The subject program was selected from a textbook [38] and could play Chinese chess (or Xiangqi) in either of two modes: play automatically on computer or play with a user of the subject program.

The subject program displays a Xiangqi board on screen for user (player) as shown in Figure 4. On the board there are 32 pieces at the beginning of game: 16 in black and 16 in red. Table 3.1 tabulates some of details of Xiangqi pieces[7].

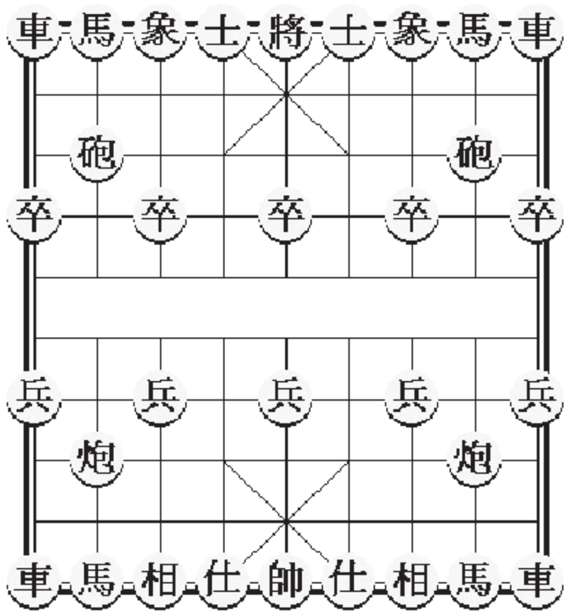

FIGURE 4
BOARD AND PIECES OF CHINESE CHESS (XIANGQI)

6 A theoretical stopping criterion is $Y_{t+1} = 0$ since the goal behind our optimal control policy (Step 3) is to remove all the defects at least expected cost. However in practice $Y_{t+1} = 0$ can seldom be satisfied since $N^{(t+1)}$ is just an estimate of N. A more practical stopping criterion is that Y_{t+1} is below some given threshold, for example.

7 Many web sites are available for Xiangqi game. See, e.g., www.chessvariants.com/xiangqi.html.

TABLE 1
DETAILS OF PIECES OF XIANGQI

Image	Name	No. on each side	Abbreviation
將 帥	General (King)	1	K
士 仕	Mandarin (Assistant)	2	A
象 相	Elephant	2	E
馬 馬	Horse	2	H
車 車	Chariot (Rook)	2	R
砲 炮	Cannon	2	C
卒 兵	Soldier (Pawn)	5	P

The subject program is supposed to correctly implement all Xiangqi rules. These rules serve as the specification of the subject program. If any move of the game violates any of the rules, then we observe a failure of the subject program and there must be (injected) defects remaining in the subject program.

10 software defects were intentionally injected into the subject program, which was in turn submitted to one tester who knew details of the injected software defects. Each of the injected software defects would lead at least one red move of pieces to break some of the Xiangqi rules, but none of the injected software defects would lead black pieces to break the Xiangqi rules. The tester served as one player in black and the computer served as the other player in red. Each move of any black piece available on the Xiangqi board made up a test case, and the automatic response of the red player served as the output of the subject program for the given test case. So, each test case would make the subject program behave properly or lead it to a failure. In order to apply the adaptive testing strategy and the random testing strategy (see Section 3.2 below), the 16 black pieces were divided into four classes:

Class 1 comprises 4 pieces: two Chariot's and two Cannon's

Class 2 comprises 5 pieces: five Soldier's on the border lines

Class 3 comprises 4 pieces: two Elephant's and two Horse's

Class 4 comprises 3 pieces: the General and two Mandarin's

Remarks:

(1). The injected defects were distributed over two sub-procedures in the subject program: ch-red and ck-red. The ch-red sub-procedure checked a look-up table, determining all potential admissible moves of each red piece. The ck-red sub-procedure picked up all admissible moves for each red piece under the current scenario of game. Each injected defect might lead to a failure individually or in conjunction with one more detected defect. The reason of injecting defects into these two sub-procedures was that failures of these sub-procedures would certainly lead to breakdown of some of the Xiangqi rules and defects in them were not easy to be triggered.

(2). The reason of dividing the input domain of the subject program into four (equivalence) classes was that the number of classes should not be too small or too large. The adaptive testing strategy might reduce to the random testing strategy if the input domain comprised only one class. However big number of classes of input domain means that there were too many parameters to be estimated on-line by the genetic algorithm at high computational complexity. In our case five parameters were estimated on-line.

(3). There was no hard justification why a piece was put into a particular class. The allocation of pieces was rather arbitrary.

(4). The defects were injected by an undergraduate student, whereas another undergraduate student served as the tester. Both undergraduate students had good understanding of the Xiangqi rules, the source code of the subject program, and the injected defects.

3.2 Test Process

The adaptive testing strategy was applied to the subject program with injected defects and the test process was stopped when all the 10 injected defects were detected and removed. Since each test case (a move of black piece) revealed at most one failure, we supposed that each test case detected at most one injected defect. Test cases were selected by the adaptive test strategy and applied to the subject program one by one. A failure was observed if the subject program broke some of the Xiangqi rules and then the tester removed the failure-causing defect. More specifically, the adaptive testing of the subject program was performed as follows:

Algorithm A

BEGIN

Step 1 Set up the operating mode of the subject program, under which the tester served as the black player and the computer served as the red player; the initial scenario of game was as shown in Figure 4.

Step 2 Selected one of the available classes of black pieces by the adaptive testing strategy;

Step 3 Picked up an available piece from the selected class randomly, following a uniform distribution;

Step 4 Found out all the admissible moves that the picked-up piece could perform, in the sense that these moves complied with the Xiangqi rules;

Step 5 Chose an admissible move randomly, following a uniform distribution;

Step 6 If the piece selected could not move according to the rule, set $Z_t = 0$; went to Step 13;

Step 7 Performed the chosen black move; the red player (computer) would respond automatically by moving a red piece;

Step 8 If the red move broke the Xiangqi rules, then a failure was observed and $Z_t = 1$ was set;

Step 9 The tester detected and removed the failure-causing defect;

Step 10 If the number of detected defects reached 10, stopped the testing process; went to Step 14;

Step 11 If none of the two sides (black or red) won, went to Step 13;

Step 12 Reset the Xiangqi game;

Step 13 Fed the test data $\{\text{selected class}, Z_t\}$ and the state S_i of every class to the adaptive testing strategy; went to Step 2;

Step 14 Stopped;

END

The random testing strategy followed *Algorithm R*, which was as same as *Algorithm A* except Step 2. In Step 2 of Algorithm R, one of the four classes of the black pieces was selected in accordance with a uniform distribution.

Remarks:

(1). In Step 3, the uniform distributions varied with sizes of the classes. For class 1, each piece had probability of $1/4$ being chosen at the beginning of game. For class 4, the corresponding probability should be $1/3$ at the beginning of game. However during the game some pieces might be removed from the board and thus the size of a class decreased accordingly. Suppose one cannon was removed from class 1, then each of the remaining three pieces had probability of $1/3$ being chosen at the beginning of game from class 1. The uniform distributions used in Step 5 should be interpreted in the same way. If a chosen piece had 9 admissible moves under current scenario of game, then each move had probability of $1/9$ being performed.

(2). Step 4 was automatically performed by an algorithm that was implemented in the test environment in conjunction with the subject program. No human intervention was allowed in the step.

(3). In Step 12 the Xiangqi game was reset to an initial scenario as shown in Figure 6.1. All pieces were available and in their initial positions.

(4). Step 12 changed the physical state of the subject program, but did not change the current number of injected defects remaining in the subject program. That is, it was Step 9, not Step 12, that changed the state of the control model of Markov chain.

(5). The test oracle used in Step 8 was implemented in the test environment. $Z_t = 0$ or $Z_t = 1$ was automatically set. No on-line human intervention was involved for the test oracle.

(6). The tester was involved in Steps 1, 9 and 14. The tester removed detected defects and stopped the testing process whenever appropriate. No other human interventions were allowed in the testing process.

(7). The testing strategy defined by Algorithm R is a special form of partition testing since it selects an equivalence class at random each time. The number of test cases selected from an equivalence class is a random variable whose realization is determined online according to Algorithm R rather than a priori. Thus the testing strategy defined by Algorithm R can more accurately be referred to as 'random-partition' testing strategy. A reason for us to adopt Algorithm R in the case study is that we really have no idea how to determine a priori the number of test cases that should be selected from an equivalence class. Accordingly, the testing strategy defined by Algorithm A is referred to as 'adaptive' testing strategy although it can more appropriately be called 'adaptive-partition' testing strategy.

3.3 Test Results

In the case study Algorithm A and Algorithm R were performed alternatively as follows: Algorithm A → Algorithm R → Algorithm A → Algorithm R → … Each of the two algorithms was performed for 10 times. Each time the same subject program with the 10 injected defects was subjected to Algorithms A and R. Table 2 tabulates the numbers of test cases used between successive observed failures. In Figure 5, the horizontal axis represents the number of detected defects, whereas the vertical axis represents the total number of tests applied from removal of the last injected defect to detection of a new injected defect in the 10 test processes under a testing strategy.

TABLE 2
TEST RESULTS OF ALGORITHMS A AND R

k : number of detected defects

$bn_A(k)$: the number of test cases used between the $(k-1)$th detected defect and the kth detected defect under Algorithm A (the adaptive testing strategy)

$bn_R(k)$: the number of test cases used between the $(k-1)$th detected defect and the kth detected defect under Algorithm R (the random testing strategy)

$$\sum(j) : \sum_{k=1}^{10} bn_A(k) \text{ or } \sum_{k=1}^{10} bn_R(k) \text{ of test process } j$$

Test Process 1 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(1)$
$bn_A(k)$	2	6	7	1	7	11	19	4	113	105	275
$bn_R(k)$	1	2	4	8	3	6	16	18	342	367	767
Test Process 2 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(2)$
$bn_A(k)$	2	2	4	2	4	6	15	30	8	82	155
$bn_R(k)$	1	1	3	43	2	22	13	201	144	112	542
Test Process 3 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(3)$

$bn_A(k)$	1	5	2	5	20	27	27	25	121	6	239
$bn_R(k)$	2	4	15	10	23	26	7	46	8	325	466
Test Process 4 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(4)$
$bn_A(k)$	2	3	3	34	10	5	10	54	45	111	277
$bn_R(k)$	3	2	1	6	6	5	41	113	91	51	319
Test Process 5 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(5)$
$bn_A(k)$	1	1	4	18	6	49	14	49	176	69	387
$bn_R(k)$	3	2	10	1	7	16	21	6	151	37	254
Test Process 6 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(6)$
$bn_A(k)$	2	4	2	2	21	43	31	31	20	86	242
$bn_R(k)$	1	4	2	11	20	35	37	37	251	405	803
Test Process 7 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(7)$
$bn_A(k)$	1	3	2	69	24	3	15	14	76	161	368
$bn_R(k)$	1	2	10	22	5	7	15	41	53	38	194
Test Process 8 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(8)$
$bn_A(k)$	1	1	2	9	22	49	27	30	27	210	378
$bn_R(k)$	2	2	6	4	22	13	17	32	150	149	397
Test Process 9 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(9)$
$bn_A(k)$	1	3	4	24	8	54	14	25	6	495	634
$bn_R(k)$	1	4	1	2	33	20	11	71	54	156	353
Test Process 10 (k)	1	2	3	4	5	6	7	8	9	10	$\sum(10)$
$bn_A(k)$	1	2	2	6	11	22	43	135	37	131	390
$bn_R(k)$	1	2	1	7	22	20	41	181	100	50	425

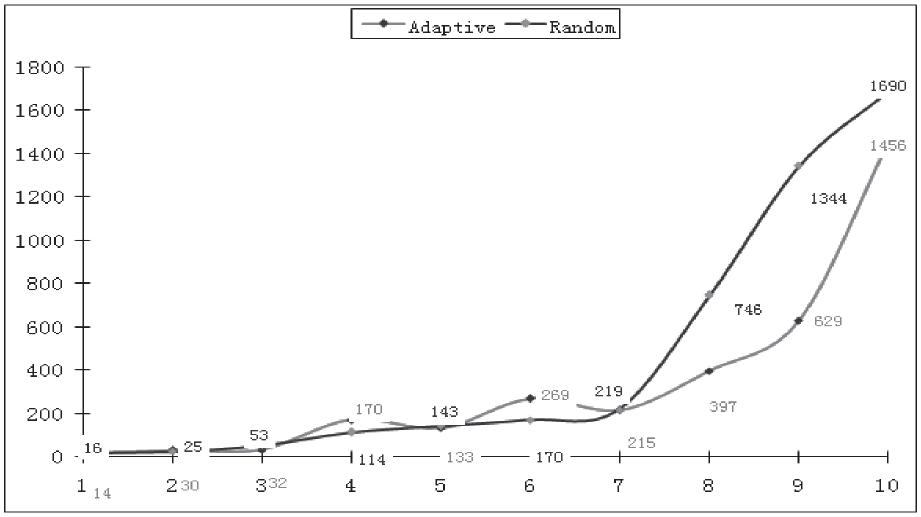

FIGURE 5
ADAPTIVE TESTING VERSUS RANDOM TESTING: TEST CASES USED TO DETECT A NEW DEFECT

Here we should note that the case study presented here is closely related to that presented in reference [10]. The two case studies used the same subject program, and were both aimed to justify the advantages of the adaptive testing strategy over the random testing strategy. The case study presented here is actually a follow-up of that presented in reference [10]. However the two case studies are different in several aspects. First, different sets of injected defects were used. In the case study presented in this section, 10 defects were injected, whereas in the case study presented in reference [10], 20 defects were injected. Second, a test case had different physical interpretation. In the case study presented in this section, a test case comprised a move of black piece. In the case study presented in reference [10], a test case comprised a move of black piece and the corresponding move of red piece. Third, the two case studies had different divisions of the input domain of the subject program, although both divisions comprised four equivalence classes. Finally, the two case studies used different stopping criteria of testing. In the case study presented in this section, software testing would not be stopped until all the 10 injected defects are removed. In the case study presented in reference [10], software testing would be stopped upon the 10th failure-causing test is observed. A single test might discover more than one defect since it comprised two moves.

4 Discussion

From the above test results we can have the following observations:

(1). In the 10 test processes, the adaptive testing strategy used 1456 test cases to observe the 10th failure (since the 9th failure), whereas the random testing strategy needed 1690 test cases. The adaptive testing strategy noticeably outperformed the random testing strategy.

(2). In the early phase of testing (before the 6th or 7th failure was observed), the random testing strategy and the adaptive testing strategy behaved similarly. However after the early phase of testing, the number of test cases used to detect one more defect by the random testing strategy grew rapidly, whereas the adaptive testing strategy demonstrated a similar but slightly different scenario. This means that in the late phase of testing, software defects are more and more difficult to be detected.

(3). In 2 (test processes 5 and 7) of the 10 test process, the random testing strategy outperformed the adaptive testing strategy, in the sense that the total number of test cases applied before the 10th failure was observed for the random testing strategy was less than that for the adaptive testing strategy. Or roughly speaking, in most cases or 80% of cases the adaptive testing strategy outperformed the random testing strategy.

(4). In test process 4, the adaptive testing strategy used 277 test cases in total to have the 10 failures observed, where this number was 319 for the random testing strategy. So we can say that the adaptive testing strategy outperformed the random testing strategy. However the adaptive testing strategy used 10 test cases from the 4th failure to the 5th failure and used 111 test cases from the 9th failure to the 10th failure, whereas these numbers for the random testing strategy were 6 and 51, respectively. So, in order to have a reasonable comparison of the adaptive testing strategy and the random testing strategy, we should not stick to a single defect or a single test process. There is no absolute guarantee that the adaptive testing strategy must outperform the random testing strategy in any scenarios. The adaptive testing strategy outperforms the random testing strategy in a statistical sense. This complies with the statistical nature of the adaptive testing strategy and the random testing strategy.

(5). In the 10 test processes, the adaptive testing strategy used 3345 test cases in total, whereas this number was 4520 for the random testing strategy. Overall, the adaptive testing strategy outperformed the random testing strategy about 26% in terms of the number of the required test cases. This is an encouraging observation.

(6). Let us consider the dispersion of test results and examine how stable a testing strategy may be. Note that for the adaptive testing strategy, we have

$$\sigma_a = \sqrt{\frac{1}{10}\sum_{i=1}^{10}\left[\sum(i) - \frac{1}{10}\left(\sum(1) + \cdots + \sum(10)\right)\right]^2} = 394.07$$

σ_a is just the standard deviation of the sample $\{\sum(1),\ldots,\sum(10)\}$ generated by the adaptive testing strategy and can be treated as a measure of how stably the adaptive testing strategy behaves. Similarly, for the random testing strategy, we have

$$\sigma_r = \sqrt{\frac{1}{10}\sum_{i=1}^{10}\left[\sum(i) - \frac{1}{10}\left(\sum(1) + \cdots + \sum(10)\right)\right]^2} = 606.41$$

Since $\left|\frac{\sigma_a - \sigma_r}{\sigma_r}\right| \times 100\% \approx 35.02\%$, we can roughly say that the behavior of the adaptive testing strategy is 35% more stable than that of the random testing strategy. This is a desirable result.

Of course we should note that there are several threats to the validity of the above observations and justifications. First, the subject program used in the case study is small. It contains only over 1,000 lines of code. Second, the 10 defects were injected into parts of code that invoke the two sub-procedures (ch-red and ck-red). No defects were injected into other parts. The effects of the nature and the number of injected defects on the test results need further investigation. Finally, the test results may depend on how the input domain of a subject program is divided.

While various threats to the validity of test results are kept in mind, we should say that the above test results and those presented in our previous works [11, 12] are encouraging. One may wonder why the CMC approach or the control-theoretic approach can help to improve the software testing processes even if there are irrationals associated with the assumptions taken in the present form of the CMC approach to software testing. We can have the following justifications.

(1). The control-theoretic approach takes full usage of available testing information or the whole history of testing on-line. This is not the case for existing approaches to software testing. The information offers a potential for the control-theoretic approach to outperform the existing approaches to software testing.

(2). Since the control-theoretic approach formalizes the feedback mechanisms in software testing, the required testing strategy is derived rigorously on a solid theoretic foundation. Available information is better used than otherwise. This may avoid some of potential human biases or errors. In this way the control-theoretic approach helps to improve software testing.

(3). Optimization criteria are used in the control-theoretic approach. This means the control-theoretic approach uses additional information in comparison with existing approach to software testing. Roughly speaking, if the existing approaches are treated as a general-purpose one, then the control-theoretic approach is a specific-purpose one. It is not surprising that the control-theoretic approach outperforms the existing approaches to software testing in terms of given optimization criteria.

(4). In practice software defects tend to cluster: a relatively few modules, in a set of modules constituting a program, will contain a disproportionate number of defects [14, 31]. This offers adaptive software testing strategy a good chance to behave well. By learning from the testing history, the adaptive testing strategy will avoid invoking various modules evenly. Rather, the adaptive software testing strategy will tend to invoke certain modules more frequently than other modules.

(5). Although the required test suite of, e.g., software system testing may be determined in the phase of software requirement analysis, how to sequence test cases is still a problem. Different sequencing of test cases will lead to different behavior of software testing process. This may be the one of major reasons for studying the test case prioritization problem that is concerned with how to select a permutation of a given test suite such that a given quantitative function (e.g., rate of defect detection) is optimized [24, 34]. A big advantage of the control-theoretic approach is that it dynamically sequences test cases on-line in an optimal manner and thus improves software testing process.

(6). As long as no debugging is involved, the main weakness of the present form of the CMC approach is that it implicitly assumes that remaining software defects are equally detectable. However we should note that different test cases have different capability of detecting software defects. The fact that remaining defects are not equally detectable becomes less important. Adaptive testing strategy tries to find test cases that are most likely to reveal failure or fit the testing goal. It is not necessarily aimed to find most detectable defects.

(7). The present form of the CMC approach completely avoids various debugging activities. It assumes that every defect is removed immediately upon being detected without introducing new defects or new functional features. In practice software debugging may be delayed to the time instant until a given cumulative number of observed failures is reached or a given schedule is met. However this does not mean that the CMC approach will not be able to accommodate debugging activities. Actually we may assume that a "virtual" debugging action is taken each time a defect is detected, but it may or may not lead to removal of the detected defect. Only those debugging activities that are taken in actual debugging processes by software developers can remove software defects. Reference [25] shows how debugging processes can be incorporated in the CMC approach. More reasonable assumptions should lead to better performance of the adaptive approach to software testing.

5 THE OPTIMAL STOPPING PROBLEM

Considerable amount of research works has been devoted to the optimal stopping problem of software testing [15, 26, 33, 37]. These works try to balance the conflicting requirements of desired reliability goals and cost constraints. The optimal stopping time is determined when an objective function involving reliability goals and cost constraints is maximized or minimized. Most of them only consider continuous-time domain, although there are some exceptions that involve discrete-time domain [28]. Further we note that software reliability behavior varies with the underlying software test or operational profile. When inconsistency between test profile and operational profile is observed, adjustments should be made to software reliability estimations [5]. Therefore the optimal stopping problem should explicitly consider the effects of the underlying software test or operational profile. Unfortunately this is not the case in the previous research works on the optimal stopping problem. To overcome this deficiency, in this section we discuss the optimal stopping problem in the setting of controlled Markov chains.

5.1 Problem Formulation

In order to discuss the optimal stopping problem, we include an additional action $\Im$ in the action set. When the action $\Im$ is taken, the software testing is stopped and the software moves to an absorbing state. For convenience, we still denote the absorbing state as $\Im$. Let

$$Y_t = \begin{cases} j & \text{if at time } t \text{ the software contains } j \text{ defects} \\ & \text{and the software testing has not been stopped}; \ j = 0,1,2,\ldots; \ t = 0,1,2,\ldots \\ \Im & \text{if at time } t \text{ the software testing has been stopped} \\ & ; t = 0,1,2,\ldots \end{cases}$$

and

$$Z_t = \begin{cases} 1 & \text{if the action taken at time } t \text{ detects a defect} \\ 0 & \text{if the action taken at time } t \text{ doesn't detect a defect} \\ & ; t = 0,1,2,\ldots \end{cases}$$

We have the following assumptions.

(1). The software contains $N \geq 1$ defects at the beginning ($t = 0$).

(2). An action at each time detects at most one defect, but action $\Im$ does not detect any defect.

(3). If a defect is detected, then it is removed immediately and no new defects are introduced; that is, $Y_t = j$ and $Z_t = 1$ mean $Y_{t+1} = j - 1$.

(4). If no defect is detected by an action, then the number of remaining software defects remains unchanged; that is, $Y_t = j(\neq \Im)$ and $Z_t = 0$ mean $Y_{t+1} = j$.

(5). $Y_t = \Im$ is a unique absorbing state; it is the target state.

(6). At every time there are always $m+1$ admissible actions; the action set is $A = \{1, 2, \ldots, m, \Im\}$. Action $\Im$ does not detect a defect.

(7). The transition probabilities of software states are independent of time. Let $q_{y_1 y_2}(a)$ denote the transition probability of the software from state y_1 to state y_2 when action a is taken. For $k \in \{1,2,\ldots,N\}, l \in \{\Im,0,1,\ldots,N\}$ and $a \in \{1,2,\ldots,m\}$ there holds

$$q_{kl}(a)=\begin{cases}1-k\theta_i & ;l=k, a=i\\ k\theta_i & ;l=k-1, a=i\\ 0 & ;\text{otherwise}\end{cases}$$

For $k=0, l \in \{\Im,0,1,\ldots,N\}$ and $a \in A$, there holds

$$q_{0l}(a)=\begin{cases}1 & ;\text{if } l=\Im, \forall a \in A\\ 0 & ;\text{if } l \neq \Im, \forall a \in A\end{cases}$$

For $k, l \in \{\Im,0,1,\ldots,N\}$ and $a=\Im$, there holds

$$q_{kl}(\Im)=\begin{cases}1 & ;\text{if } l=\Im\\ 0 & ;\text{otherwise}\end{cases}$$

(8). Action A_t taken at time t incurs a cost of $W_{Y_t}(A_t)$, no matter whether or not it detects a defect;

$$W_{Y_t}(a)=\begin{cases}w_j(a)>0 & \text{if } Y_t=j, (j=1,2,\ldots,N) \quad \forall a \in A\\ 0 & \text{if } Y_t=0 \text{ or } \Im, \quad \forall a \in A\end{cases}$$

(9). The cost of removing a detected defect is ignored.

Let τ be the first-passage time to state $\Im$, and

$$J_\omega(N)=E_\omega\sum_{t=0}^{\tau}W_{Y_t}(A_t)$$

where ω denotes a control policy (testing strategy) and A_t the action taken at time t. Our problem is to find the optimal control policy (testing strategy) that minimizes $J_\omega(N)$. Figure 5.1 depicts the software state transition diagram.

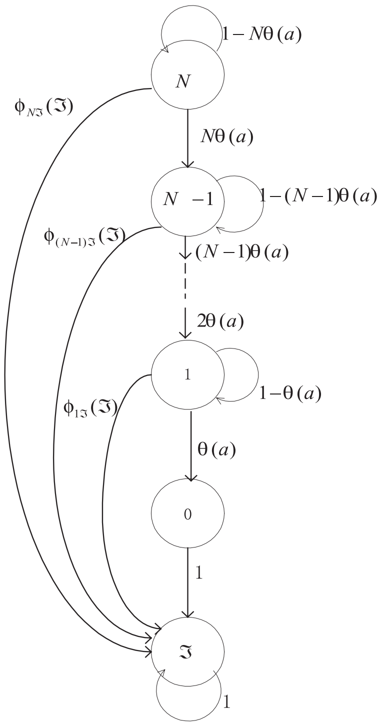

FIGURE 6
STATE TRANSITION DIAGRAM OF MULTIPLE-DEFECTS SOFTWARE WITH STOPPING STATE

We note that the mathematical framework presented here is similar to that presented in Section 2 and also defines a controlled Markov chain. Although we treat state $\Im$ as a unique absorbing state and do not treat state 0 as another absorbing state, there should be no effects on the optimal action that should be taken in state $j \geq 1$ as a result of $w_0(a) \equiv 0, \forall a \in A$.

We also note that the assumption $w_j(\Im)>0$; $j=1,2,\ldots,N$ is essential. If the software still contains j defects and the testing process is stopped, then the j defects are left to future detection or failure that may incur much higher costs. If $w_j(\Im)=0$ holds, then $J_\omega(j)=0$ and we do not need to conduct software testing any more.

5.2 The CMC Approach

To determine the optimal policy (testing strategy), we still follow the method of successive approximation in the setting of controlled Markov chains. Let

$$v_{n+1}(0) = \min_{a \in A}\{w_0(a) + q_0(a)v_n(0)\} \equiv 0$$

$$v_{n+1}(j) = \min_{a \in A}\left\{w_j(a) + \sum_{k \neq \Im} q_k(a)v_n(k)\right\}$$

with arbitrary $v_0(k)$, $j = 1,2,\ldots,N$ (5.1)

Proposition 5.1 It holds

$$v(1) = \lim_{n\to\infty} v_n(1) = \min\left\{w_1(\Im), \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}\right\} \quad (5.2)$$

Proof See Appendix 4. Q.E.D.

Equation (5.2) is just to say that the software that contains one defect should not be tested at all if $w_1(\Im) \le \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}$. Otherwise the optimal action that minimizes $\frac{w_1(i)}{\theta_i}$ with respect to i should be taken and the software continues to be under test till the software defect is detected and removed.

Proposition 5.2 It holds

$$v(j) = \min\left\{w_j(\Im), \min_{1\le i\le m}\left\{\frac{w_j(i)}{j\theta_i} + v(j-1)\right\}\right\} \quad ; j = 2,3,\ldots,N \quad (5.3)$$

Proof See Appendix 5. Q.E.D.

Propositions 5.1 and 5.2 give a clear picture of how to test software or when to stop testing. If $w_j(\Im) \equiv \infty,\ j = 1,2,\ldots,N$, that is, the action $\Im$ is not admissible, then in state j the optimal action is the one that minimizes $\frac{w_j(i)}{\theta_i}$ with respect to i and is independent of $v(j-1)$. This coincides with the conclusion presented in Section 2. If $w_j(\Im) < \infty$, or $\Im$ is admissible, then the optimal action that should be taken in state j depends not only on $\frac{w_j(i)}{\theta_i}$, but also on $v(j-1)$ and $w_j(\Im)$. Note that $\min_{1\le i\le m}\left\{\frac{w_j(i)}{j\theta_i} + v(j-1)\right\}$ is the minimal cost when $\Im$ is not admissible. To decide which action should be taken, we just compare the action $\Im$ with the optimal action under the assumption that $\Im$ is not admissible.

Further if $w_j(a)$ is independent of j, or $w_j(a) \equiv w(a)$, then in each state j, $(j = 1,2,\ldots,N)$, there are two possible actions that can be taken: the one that minimizes $\frac{w(i)}{\theta_i}$ with respect to i (suppose the extrema is unique) and $\Im$.

Example 5.1 Suppose

$$N = 2,\ A = \{1, 2, \Im\}, \theta_1 = 0.1, \theta_2 = 0.2$$

$$w_1(1) = 20,\ w_1(2) = 30,\ w_1(\Im) = 100$$

$$w_2(1) = 15,\ w_2(2) = 25,\ w_2(\Im) = 240$$

Then from equation (4.3) we arrive at

$$v(1) = \min\left\{100, \frac{20}{0.1}, \frac{30}{0.2}\right\} = 100$$

$$v(2) = \min\left\{240, \frac{15}{2\times 0.1} + 100, \frac{25}{2\times 0.2} + 100\right\} = 162.5$$

This suggests that at the beginning (when the software under test contains two defects), action 2 should be taken until a defect is detected and removed. After a defect is removed, the software testing should be stopped.

6 Concluding remarks

Software testing is arguably the least understood part of software development process. In comparison with existing approaches to software testing such as white-box approach, black-box approach and random approach, the CMC approach treats software testing as a control problem and does not stick to particular techniques and is basically process oriented. The framework of the CMC approach is rather general and the concept of an action can be interpreted widely. An action can be a specific testing technique, a software run, a test case, an equivalence class of test cases, and so forth. An optimization criterion involving software defects and testing costs is given explicitly and a priori, and the resulting control policy (testing strategy) is derived rigorously. By incorporating a parameter estimation component into the CMC approach, an adaptive testing strategy can be obtained and serves as the counterpart of adaptive control in software testing. In the preceding sections we revisit the CMC approach, present a new case study of adaptive testing and discuss why the CMC approach or control-theoretic approaches to software testing can work in practice. We also show that the CMC approach can also be applied to the optimal stopping problem of software testing. This further justifies the effectiveness of the CMC approach in particular, and that of the idea of software cybernetics in general.

References

[1] A. Arapostathis, V.S. Borkar, E. Fernandez-Gaucherand, M.K. Ghosh, S.I. Marcus, "Discrete-Time Controlled Markov Processes with Average Cost Criterion: A Survey," *SIAM Journal on Control and Optimization*, vol.31, no.2, 282-344, 1993.

[2] B. Beizer, "*Software Testing Techniques (2nd Edition)*," Van Nostrand Reinhold, 1990.

[3] R.E. Bellman, S.E. Dreyfus, "*A lied Dynamic Programming*," Princeton University Press, 1962.

[4] R.V. Binder, "*Testing Object-Oriented Systems: Models, Patterns, and Tools,*" Addison-Wesley, 2000.

[5] K.Y. Cai., "*Software Defect and Operational Profile Modeling*," Kluwer Academic Publishers, 1998.

[6] K.Y. Cai, "Towards a Conceptual Framework of Software Run Reliability Modeling," *Information Sciences*, vol.126, 137-163, 2000.

[7] K.Y. Cai, "Optimal Software Testing and Adaptive Software Testing in the Context of Software Cybernetics," *Information and Software Technology*, vol. 44, 841-855, 2002.

[8] K.Y. Cai, J.W. Cangussu, R.A. DeCarlo, A.P. Mathur, "An Overview of Software Cybernetics," *Proc. the 11th International Workshop on Software Technology and Engineering Practice*, IEEE Computer Society Press, 77-86, 2004.

[9] K.Y. Cai, T.Y. Chen, T.H. Tse, "Towards Research on Software Cybernetics," *Proc. 7th IEEE International Symposium on High Assurance Systems Engineering,*, 240-241, 2002.

[10] K.Y .Cai, B. Gu, H.Hu, Y.C. Li, "A Case Study of Adaptive Software Testing," working paper, 2003.

[11] K.Y. Cai, B. Gu, H.Hu, Y.C. Li, "Adaptive Software Testing with Fixed-Memory Feedback," *Journal of Systems and Software*, vol. 80, 1328-1348, 2007.

[12] K.Y. Cai, C.H. Jiang, H. Hai, C.G. Bai, "An Experimental Study of Adaptive Testing for Software Reliability Assessment," *Journal of Systems and Software*, vol. 81, 1406-1429, 2008.

[13] K.Y. Cai, Y.C. Li, K. Liu, "Optimal and Adaptive Testing for Software Reliability Assessment," *Information and Software Technology*, vol. 46, 989-1000, 2004.

[14] F.T. Chan, T.Y. Chen, T.H. Tse, "On the Effectiveness of Test Case Allocation Schemes in Partition Testing," *Information and Software Technology*, vol.39, 719-726, 1997.

[15] S.R. Dalal, A.A. McIntosh, "When to Stop Testing for Large Software Systems with Changing Code," *IEEE Transactions on Software Engineering*, vol.20, No.4, 318-323, 1994.

[16] C. Derman, "*Finite State Markovian Decision Processes*," Academic Press, 1970.

[17] J.E. Duran, S.C .Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, vol. SE-10, No.4, 438-444, 1984.

[18] M.J. Escalona, J.J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, F.J. Domínguez, "An Overview on Test Generation from Functional Requirements," *Journal of Systems and Software*, vol. 84, no.8, 1379-1393, 2011.

[19] P.G. Frankl, E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering*, vol.19, No.2, 202-213, 1993.

[20] J.B. Goodenough, S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, vol.SE-3, No.2, 156-173, 1975.

[21] W.J.Gutjahr, "Partition Testing vs. Random Testing: The Influence of Uncertainty," *IEEE Transactions on Software Engineering*, vol.25, No.5, 661-674, 1999.

[22] D.Hamlet, R.Taylor, "Partition Testing Does Not Inspire Confidence*", IEEE Transactions on Software Engineering*, vol.16, No.12, 1402-1411, 1990.

[23] Hernandez-Lerma, "*Adaptive Markov Control Processes*, " Springer-Verlag, 1989.

[24] Hernandez-Lerma, J.B.Lasserre, "*Discrete-Time Markov Control Processes: Basic Optimality Criteria*," Springer, 1996.

[25] H. Hu, C.H. Jiang, K.Y. Cai, "An Improved A roach to Adaptive Testing, " *International Journal of Software Engineering and Knowledge Engineering*, vol.19, No.5, 679-705, 2009.

[26] Z. Jelinski, P.B. Moranda, "Software Reliability Research," in: W.Greiberger (ed.), *Statistical Computer Performance Evaluation*, Academic Press, 464-484, 1972.

[27] P.R. Kumar, "A Survey of Some Results in Stochastic Adaptive Control," *SIAM Journal on Control and Optimization*, vol.23, no.3, 329-380, 1985.

[28] B. Littlewood, D. Wright, "Some Conservative Sto ing Rules for the Operational Testing of Safety-Critical Software," *IEEE Transactions on Software Engineering*, vol.23, no.11, 673-683, 1997.

[29] S. Mouchawrab, L.C. Briand, Y. Labiche, M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments, " *IEEE Transactions on Software Engineering*, vol. 37, no.2, 161-187, 2011.

[30] J.D. Musa, "Software-Reliability-Engineered Testing,"*Computer*, vol. 29, No.11, 61-68, 1996.

[31] A.S. Parrish, S.H. Zweben, "On the Relationships among the All-uses, All-DU-paths, and All-edges Testing Criteria," *IEEE Transactions on Software Engineering*, vol.21, no.12, 1006-1010, 1995.

[32] M.L. Puterman, "*Markov Decision Processes: Discrete Stochastic Dynamic Programming*," John Wiley & Sons, 1994.

[33] N.D. Singpurwalla, "Determining an Optimal Time Interval for Testing and Debugging Software," *IEEE Transactions on Software Engineering*, vol. 17, no.4, 1991, 313-319.

[34] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Transactions on Software Engineering*, vol.-12, no.12, 1128-1138, 1986.

[35] J.A. Whittaker, "What is Software Testing? And Why is it so Hard?," *IEEE Software*, January/February, 70-79, 2000.

[36] J.A. Whittaker, M.G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, vol.20, no.10, 812-824, 1994.

[37] M. Xie, "*Software Reliability Modeling*," World Scientific, 1991.

[38] C.J. Yang, "*Classical Programs in Visual Basic* (in Chinese)," Tsinghua University Press, Beijing, 2001.

[39] H. Zhu, P.A. Hall, J.H.R. May, "Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no.4, 366-427, 1997.

APPENDIX 1: CONTROLLED MARKOV CHAINS

In the setting of controlled Markov chains, the controlled object is given as a discrete-time Markov control model that is five-tuple

$$\left\langle S, A, \{A(y) | y \in S\}, Q, W \right\rangle \tag{A1.1}$$

where S and A are given sets, called the state space and the control (action) set, respectively. $\{A(y) | y \in S\}$ is a family of nonempty subsets $A(y)$ of A, with $A(y)$ being the set of feasible controls or actions in the state $y \in S$. Q is a transition law such that

$$Q(B | y, a) = \Pr\{Y_{t+1} \in B | Y_t = y, A_t = a\} \quad , B \subset S$$

where Y_t denotes the system state at time t, and A_t the action (control) applied at time t. Finally, W is a cost-per-stage (or one-stage cost) function defined on A. Usually, W is real valued and greater than zero. It represents what cost is incurred when an action is taken. When S is discrete, model (A1.1) is the so-called controlled Markov chain. An alternative title for controlled Markov chain is Markov decision process.

Control model (A1.1) represents a controlled stochastic system that is observed at times $t = 0,1,2,\ldots$. If the system is in the state $Y_t = y \in S$ at time t and the control $A_t = a \in A(y)$ is applied, then two things happen: (1). A cost $W_{Y_t}(a)$ is incurred, and (2). The system moves to the next state Y_{t+1} according to the transition law Q. Once the transition into the new state has occurred, a new control is chosen and the process is repeated. A controlled Markov chain or discrete-time Markov control model is not a Markov chain in general. It reduces to an ordinary Markov chain when A consists of only one action and $A(y) \equiv A$.

Let

$$H_t = \{Y_0, A_0, Y_1, A_1, \ldots, Y_{t-1}, A_{t-1}, Y_t\}$$

That is, H_t denotes the history of the system up to time t. Let

$$D_t(A_t; H_t) = \Pr\{A_t | H_t\}$$

i.e., $D_t(a; h_t)$ denotes the conditional probability of $A_t = a$ at time t under the condition[8] $H_t = h_t$. Obviously,

$$\sum_{a \in A(y_t)} D_t(a; h_t) = 1$$

Then a control policy is defined as

$$\omega = \{D_t(A_t; H_t); t = 0,1,2,\ldots\}$$

If D_t is always confined to 1 or 0, that is, at any time a particular action is certainly taken, then ω is deterministic[9]. ω is randomized if it is not deterministic. If $D_t(A_t; H_t)$ depends only on t, Y_t and A_t,

8 In the context of software testing we can treat D_t as a moving or time-varying test profile at time t.

9 Inconsistent uses of terminology can be observed in the literature [16, 24]. In reference [16] 'deterministic' actually means 'stationary deterministic' here.

then the corresponding control policy is Markovian. If $D_t(A_t;H_t)$ depends only on Y_t and A_t, then the corresponding control policy is stationary Markovian (stationary for brevity). In this way we have several classes of control policies: randomized policy, (randomized) Markov policy, (randomized) stationary policy, deterministic policy, deterministic Markov policy, and deterministic stationary policy. Under stationary control policies, we can easily see that the resulting $\{Y_t;t=0,1,2,\ldots\}$ forms a time-homogenous Markov chain, i.e., one whose transition probabilities don't change with time. But this is not true under a general control policy.

Let $W_{Y_t}(A_t)$ denote the cost incurred at time t by action A_t applied to the system being in state Y_t. Obviously $W_{Y_t}(A_t)$ is a random variable in general. In the optimal control of a controlled Markov chain we need to optimize a performance criterion involving $W_{Y_t}(A_t)$. In the literature [16, 24], several classes of performance criteria are proposed, including average cost criteria, first-passage cost criteria, and discounted cost criteria, among others. Here we are most interested in the first-passage cost criteria or the first-passage problem. Suppose the system begins with state $Y_0=y_0$ and has a target state $y^*\neq y_0$ where the system evolution process should be stopped. Let τ denote the smallest positive integer such that $Y_\tau=y^*$. The optimal first-passage problem is concerned with minimizing the objective function

$$J_\omega(y_0)=E_\omega\sum_{t=0}^{\tau}W_{Y_t}(A_t)=\sum_{t=0}^{\tau}\sum_y\sum_a \Pr^{(\omega)}\{Y_t=y,A_t=a\}W_y(a)$$

with respect to control policy ω, where E_ω denotes the mathematical expectation under control policy ω, $\Pr^{(\omega)}$ denotes the corresponding probability under control policy ω. Since the outcome of every action is uncertain (obeying the transition law Q), τ is a random variable and therefore $J_\omega(y_0)$ is the expected value of a random sum of random variables. In this way, $J_\omega(y_0)$ represents the average cost of all paths from y_0 to state $y^*\neq y_0$ without a second visit to the state $y^*\neq y_0$.

The following lemma is basic in the theory of controlled Markov chains [16]. It asserts that the solution to the optimal first-passage problem is a deterministic stationary control policy.

Lemma A1.1 (1). If $\{W_{Y_t}(A_t),t=0,1,2,\ldots\}$ are nonnegative, then there exists a deterministic stationary control policy ω^* such that

$$J_{\omega^*}(y_0)=\inf_\omega J_\omega(y_0)$$

(2). In the case $\{W_{Y_t}(A_t),t=0,1,2,\ldots\}$ are not nonnegative, the conclusion is still valid if for any arbitrarily given initial state y_0 and control policy ω, there always holds

$$\Pr\{Y_t=y^* \text{ for some } t\geq 1 | Y_0=y_0\}=1$$

(3). The conclusions of (1) and (2) still hold if $\inf$ is replaced by $\sup$. ▮

Now the problem is how to compute the optimal control policy. Using the so-called method of successive approximation as follows can do this. Let $\{v_0(y), y\in S-\{y^*\}\}$ be arbitrary, and define

$$v_{n+1}(y)=\min_a\left\{W_y(a)+\sum_{x\in\{S-\{y^*\}\}}q_{yx}(a)v_n(x)\right\},\qquad y\in S-\{y^*\} \tag{A1.2}$$

where $q_{yx}(a)$ denotes the probability that the system is moved from state y to state x by action a. Then we have the following lemma [16].

Lemma A1.2 The optimal control policy ω^* for the first-passage problem is determined by

$$J_{\omega^*}(y_0)=\lim_{n\to\infty}v_{n+1}(y_0) \quad \text{for arbitrary } \{v_0(y), y\in S-\{y^*\}\}$$

▮

Up to this point we have reviewed several basic definitions and results of controlled Markov chains, which are required in the rest of the paper. The research of controlled Markov chains originates from Bellman's pioneering work on dynamical programming [3] and has extensively been carried out in the control and decision communities. An abundant set of literature is available. See, e.g., references [1, 3, 16, 22, 23, 26, 30] and references therein.

APPENDIX 2: CONVENTIONAL STATISTICAL APPROACH TO SOFTWARE TESTING

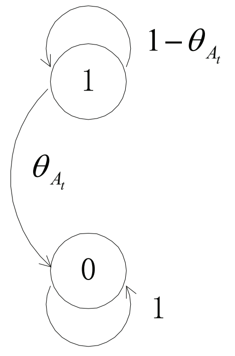

FIGURE A2.1
STATE TRANSITION DIAGRAM OF SOFTWARE UNDER TEST THAT CONTAINS ONE DEFECT AT THE BEGINNING

Let us model the testing process in the conventional statistical setting. In doing so we need to confine ourselves to the case of $N=1$ and stationary control policies (testing strategies) and try to obtain a closed-form expression of $J_\omega(1)$. In this case Figure 2.1 reduces to Figure A2.1. If action A_t (e.g., $A_t = i$) is applied to the software in state 1, then the software remains in state 1 with probability $1-\theta_{A_t}$ (e.g., $1-\theta_i$), and transitions to state 0 with probability θ_{A_t} (e.g., θ_i). Once the software enters state 0, it always remains in the state. Under a stationary control policy (testing strategy), at any time t, $D_t(A_t;H_t)$ depends only on Y_t and A_t. Since Y_t is either equal to 1 or 0, and $Y_t = 0$ is an absorbing and the target state (i.e., we don't need to consider the testing strategy for $Y_t = 0$), we can say that under a stationary policy $D_t(A_t;H_t)$ depends only on A_t. Thus we can write

$$D_t(A_t;H_t) \equiv p_i \qquad \text{if } A_t = i; \quad i = 1,2,\ldots,m$$

$$\sum_{i=1}^{m} p_i = 1$$

Therefore the stationary policy defines a time-homogenous test profile $\{\langle i, p_i\rangle; i=1,2,\ldots,m\}$ such that action i is taken with probability p_i always. The question we need to answer is: what test profile does minimize $J_\omega(1)$?

Proposition A2.1 Under assumptions (1) to (9) of Section 2 and the stationary control policy (testing strategy) $\omega = \{\{\langle i, p_i\rangle; i=1,2,\ldots,m\}, \text{at every time } t = 0,1,2,\ldots\}$, the expected cost of testing the software (removing the software defect) is determined by

$$J_\omega(1) = \frac{\sum_{i=1}^{m} w(i)p_i}{\sum_{i=1}^{m} \theta_i p_i} \tag{A2.1}$$

with

$$\max_\omega J_\omega(1) = \max_i \left\{ \frac{w(i)}{\theta_i} \right\}$$

$$\min_\omega J_\omega(1) = \min_i \left\{ \frac{w(i)}{\theta_i} \right\}$$

The optimal control policy (testing strategy) is deterministic, given by distribution $\langle p_1, p_2, \ldots, p_m\rangle$ with

$$p_i = \begin{cases} 1 & \text{if } J_\omega(1) = \dfrac{w(i)}{\theta_i} \\ 0 & \text{otherwise} \end{cases}$$

Proof Let τ be the smallest integer such that $Y_\tau = 0$. Then $\tau = l$ iff $Z_0 = Z_1 = \cdots = Z_{l-1} = 0$ and $Z_l = 1$. Immediately,

$$\Pr\{\tau = l\} = \left(\sum_{k=1}^{m} (1-\theta_k)p_k \right)^l \sum_{k=1}^{m} \theta_k p_k$$

For brevity we write $W_{Y_t}(A_t)$ as W_t and E_ω as E. We have

$$J_\omega(1) = E\sum_{t=0}^{\tau} W_t = \sum_{l=0}^{\infty} E\{W_0 + W_1 + \cdots + W_l | \tau = l\} \Pr\{\tau = l\}$$

Note

$$\begin{aligned} & E\{W_0 + W_1 + \ldots + W_l | \tau = l\} \\ & = E\{W_0|\tau = l\} + E\{W_1|\tau = l\} + \cdots + E\{W_{l-1}|\tau = l\} + E\{W_l|\tau = l\} \\ & = E\{W_0|Z_0 = 0\} + E\{W_1|Z_1 = 0\} + \cdots + E\{W_{l-1}|Z_{l-1} = 0\} + E\{W_l|Z_l = 1\} \end{aligned}$$

Further,

$$\begin{aligned} E\{W_t|Z_t = 0\} &= \sum_{i=1}^{m} w(i)\Pr\{A_t = i|Z_t = 0\} \\ &= \sum_{i=1}^{m} w(i)\frac{\Pr\{A_t = i, Z_t = 0\}}{\Pr\{Z_t = 0\}} \\ &= \sum_{i=1}^{m} w(i)\frac{\Pr\{Z_t = 0|A_t = i\}\Pr\{A_t = i\}}{\sum_{k=1}^{m}\Pr\{Z_t = 0|A_t = k\}\Pr\{A_t = k\}} \\ &= \sum_{i=1}^{m} w(i)\frac{(1-\theta_i)p_i}{\sum_{k=1}^{m}(1-\theta_k)p_k} \qquad ; t = 0,1,\ldots,l-1 \text{ with } Y_t \equiv 1 \end{aligned}$$

Similarly,

$$E\{W_l|Z_l = 1\} = \sum_{i=1}^{m} w(i)\frac{\theta_i p_i}{\sum_{k=1}^{m}\theta_k p_k}$$

Hence

$$J_\omega(1) = \sum_{l=0}^{\infty}\left[l \frac{\sum_{i=1}^{m} w(i)(1-\theta_i)p_i}{\sum_{k=1}^{m}(1-\theta_k)p_k} + \frac{\sum_{i=1}^{m} w(i)\theta_i\, p_i}{\sum_{k=1}^{m}\theta_k\, p_k}\right] \text{Ᵽ}\,\{\tau = l\}$$

$$= \sum_{l=0}^{\infty} l\left(\sum_{k=1}^{m}(1-\theta_k)p_k\right)^{l}\left(\frac{\sum_{i=1}^{m} w(i)(1-\theta_i)p_i}{\sum_{k=1}^{m}(1-\theta_k)p_k}\right)\sum_{k=1}^{m}\theta_k p_k + \sum_{l=0}^{\infty}\left(\sum_{k=1}^{m}(1-\theta_k)p_k\right)^{l}\sum_{i=1}^{m} w(i)\theta_i p_i$$

Since for $0 < a < 1$, there hold

$$\sum_{l=0}^{\infty} a^l = \frac{1}{1-a}$$

$$\sum_{l=0}^{\infty} l\cdot a^l = \frac{a}{(1-a)^2}$$

Therefore

$$J_\omega(1) = \left(\sum_{k=1}^{m}\theta_k p_k\right)\cdot\frac{\sum_{i=1}^{m} w(i)(1-\theta_i)p_i}{\sum_{k=1}^{m}(1-\theta_k)p_k}\cdot\frac{\sum_{k=1}^{m}(1-\theta_k)p_k}{\left(1-\sum_{k=1}^{m}(1-\theta_k)p_k\right)^2}$$

$$+\left(\sum_{i=1}^{m} w(i)\theta_i p_i\right)\cdot\frac{1}{1-\sum_{k=1}^{m}(1-\theta_k)p_k}$$

$$= \frac{\sum_{i=1}^{m} w(i)p_i}{\sum_{i=1}^{m}\theta_i\, p_i}$$

Now we can determine the optimal control policy (testing strategy), or distribution $\langle p_1, p_2, \ldots, p_m\rangle$ that minimizes $J_\omega(1)$. Note

$$\hbar\ J_\omega(1) = \hbar\ \sum_{i=1}^{m} w(i)p_i - \hbar\ \sum_{i=1}^{m}\theta_i p_i$$

$$\frac{\partial\, \mathrm{h}\ J_\omega(1)}{\partial p_i} = \frac{w(i)}{\sum_{i=1}^{m} w(i)p_i} - \frac{\theta_i}{\sum_{i=1}^{m}\theta_i p_i} = \frac{\sum_{j\neq i}\left(w(i)\theta_j - w(j)\theta_i\right)p_j}{\left(\sum_{i=1}^{m} w(i)p_i\right)\left(\sum_{i=1}^{m}\theta_i p_i\right)}$$

So the sign of $\frac{\partial\, \mathrm{h}\ J_\omega(1)}{\partial p_i}$ is independent of p_i. This implies that $J_\omega(1)$ is a monotone (increasing or decreasing) function of p_i. Therefore, $J_\omega(1)$ achieves its extrema at point $p_i = 0$ or 1 (since $0 \le p_i \le 1$). To determine the maximum or minimum of $J_\omega(1)$, we just check these points $\left\{p_i = 0 \text{ or } 1, \sum_{i=1}^{m} p_i = 1\right\}$. If $m = 1$, then $p_1 \equiv 1$. If $m > 1$, then there are m points we need to check: $\left\{\left\langle p_i = 1,\ p_j \equiv 0,\ (j \neq i)\right\rangle, i = 1, 2, \ldots, m\right\}$. In this way,

$$J_\omega(1) = \frac{w(i)}{\theta_i} \quad \text{if} \quad p_i = 1$$

In summary,

$$\max_\omega J_\omega(1) = \max_i\left\{\frac{w(i)}{\theta_i}\right\}$$

$$\min_\omega J_\omega(1) = \min_i\left\{\frac{w(i)}{\theta_i}\right\}$$

Q.E.D.

Here we note that $\sum_{i=1}^{m} w(i)p_i$ represents the average cost of an action (test case) and $\sum_{i=1}^{m}\theta_i\, p_i$ represents the mean number of actions (test cases) for detecting the defect remaining in the software. The optimal control policy (testing strategy) that minimizes $J_\omega(1)$ specifies that at every time an identical action that minimizes $\left\{\frac{w(i)}{\theta_i}\right\}$ with respect to i should always be applied.

APPENDIX 3: THE CMC APPROACH IN THE CASE OF $N = 1$

The mathematical treatment presented in Appendix 2 looks elegant and the corresponding conclusion is simple and gives a guideline of how to choose test actions. However the conventional statistical approach suffers from obvious limitations. First, the conclusion drawn in Appendix 2 is based on the assumption that the possible control policy (testing strategy) is stationary. The approach does not answer whether the optimal control policy is certainly deterministic and stationary if all the possible control policies (including non-stationary control policies) are considered.

Second, the approach tries to obtain a closed-form expression of $J_\omega(1)$. Although a closed-form expression of the performance index of concern may be available for some special cases of software testing (e.g., testing one-defect software), it may not be available for more general cases of software testing. To alleviate these limitations, in this Appendix we apply the CMC approach.

Actually, from Section 2 we see that the software under test defines a simple controlled Markov chain. Figure 2.1 explains this graphically. Then from Appendix 1 the optimal control policy (testing strategy) that minimizes $J_\omega(N)$ must be deterministic stationary. The action to be (deterministically) chosen is a function of the state reached by the software, not of the time. In order to compute the optimal control policy (testing strategy), we can follow the method of successive approximation as described in Appendix 1.

Define

$$v_{n+1}(1) = \min_{1\le i\le m}\{w(i) + (1-\theta_i)v_n(1)\} \qquad ; n = 0,1,2,\ldots$$

for arbitrary $v_0(1)$. Let

$$\lim_{n\to\infty} v_n(1) = v$$

Then[10]

(A3.1)

In this way

$$v \le w(i) + (1-\theta_i)v \qquad ; i = 1,2,\ldots,m$$
$$v \le \frac{w(i)}{\theta_i}$$

Therefore in order to make equation (A2.1) hold, there must be

$$v = \min_{1\le i\le m}\left\{\frac{w(i)}{\theta_i}\right\} \tag{A3.2}$$

This suggests that the optimal action taken at every time must be the one that minimizes $\left\{\frac{w(i)}{\theta_i}\right\}$ with respect to i. The optimal control policy (testing strategy) is thus obtained.

Similarly, if we are interested in the worst control policy (testing strategy), we can define

$$v_{n+1}(1) = \max_{1\le i\le m}\{w(i) + (1-\theta_i)v_n(1)\} \qquad ; n = 0,1,2,\ldots$$

for arbitrary $v_0(1)$, and conclude that the worst control policy (testing strategy) takes the action that maximizes $\left\{\frac{w(i)}{\theta_i}\right\}$ with respect to i at every time.

A direct consequence of the above conclusions is that neither random testing strategies nor partition testing strategies are optimal with respect to $J_\omega(1)$. A random testing strategy means that an action is randomly chosen each time and thus corresponds to a non-deterministic control policy (refer to Appendix 1). A partition testing strategy chooses each possible action at least once. In order to detect and remove the software defect at the least expected cost, we should devote all the testing resources to the action that minimizes $\left\{\frac{w(i)}{\theta_i}\right\}$ with respect to i. However random testing and partition testing are also 'safe' in the sense that they are not the worst strategy that maximizes the expected cost of detecting and removing the software defect. Therefore, if we are not sure which action is optimal, we should avoid devoting all the testing resources to a single action and follow a random testing or partition testing strategy.

Appendix 4: Proof of Proposition 5.1

For brevity, denote $v = v(1)$. First we have

$$\begin{aligned} v_{n+1}(1) &= \min_{a\in A}\{w_1(a) + q_1(a)v_n(1)\} \\ &= \min\left\{w_1(\Im), \min_{1\le i\le m}\{w_1(i) + (1-\theta_i)v_n(1)\}\right\} \end{aligned} \quad \text{with arbitrary } v_0(1) \tag{A4.1}$$

This means

$$v \le w_1(\Im)$$
$$v \le \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}$$

Or

$$v \le \min\left\{w_1(\Im), \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}\right\}$$

10 Note $\min(x,y)$ is a continuous function of x or y.

We consider two cases.

Case 1. Suppose

$$w_1(\Im) \le \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}$$

Then

$$w_1(\Im) \le \frac{w_1(i)}{\theta_1}$$

$$w_1(i) + (1-\theta_i)w_1(\Im) \ge w_1(\Im)$$

Therefore

$$\min_{1\le i\le m}\{w_1(i) + (1-\theta_i)w_1(\Im)\} \ge w_1(\Im)$$

and

Comparing this equation with equation (A4.1), we arrive at

$$v = w_1(\Im) = \min\left\{w_1(\Im), \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}\right\}$$

Case 2. Suppose

$$w_1(\Im) > \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}$$

From equations (A3.1) and (A3.2) we note

$$\min_{1\le i\le m}\left\{w_1(i) + (1-\theta_i)\cdot\min_{1\le k\le m}\left\{\frac{w_1(k)}{\theta_k}\right\}\right\} \le \min_{1\le k\le m}\left\{\frac{w_1(k)}{\theta_k}\right\}$$

Therefore

$$\min\left\{w_1(\Im), \min_{1\le i\le m}\left\{w_1(i) + (1-\theta_i)\cdot\min_{1\le k\le m}\left\{\frac{w_1(k)}{\theta_k}\right\}\right\}\right\}$$
$$= \min\left\{w_1(\Im), \min_{1\le k\le m}\left\{\frac{w_1(k)}{\theta_k}\right\}\right\} = \min_{1\le k\le m}\left\{\frac{w_1(k)}{\theta_k}\right\}$$

Comparing this equation with equation (A4.1) we arrive at

$$v = \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\} = \min\left\{w_1(\Im), \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}\right\}$$

Q.E.D.

APPENDIX 5: PROOF OF PROPOSITION 5.2

From equation (5.1) we arrive at

$$v_{n+1}(1) = \min_{a\in A}\{w_1(a) + q_{11}(a)v_n(1) + q_{10}(a)v_n(0)\}$$
$$= \min_{a\in A}\{w_1(a) + q_{11}(a)v_n(1)\}$$
$$= \min\left(w_1(\Im), \min_{1\le i\le m}\{w_1(i) + (1-\theta_i)v_n(1)\}\right)$$

Then from Proposition 5.1 we have

$$v(1) = \lim_{n\to\infty} v_n(1) = \min\left\{w_1(\Im), \min_{1\le i\le m}\left\{\frac{w_1(i)}{\theta_i}\right\}\right\}$$

For $j = 2$, we note

$$v_{n+1}(2) = \min_{a\in A}\{w_2(a) + q_2(a)v_n(2) + q_2(a)v_n(1) + q_0(a)v_n(0)\}$$
$$= \min\left\{w_2(\Im), \min_{1\le i\le m}\{w_2(i) + (1-2\theta_i)v_n(i)\} + 2\theta_i v_n(1)\right\}$$

In this way

$$v(2) = \min\left\{w_2(\Im), \min_{1\le i\le m}\{w_2(i) + (1-2\theta_i)v(2) + 2\theta_i v(1)\}\right\}$$

Or

$$v(2) - v(1) = \min\left\{w_2(\Im) - v(1), \min_{1\le i\le m}\{w_2(i) + (1-2\theta_i)(v(2) - v(1))\}\right\}$$

By using Proposition 5.1, we obtain

$$v(2) - v(1) = \min\left\{w_2(\Im) - v(1), \min_{1\le i\le m}\left\{\frac{w_2(i)}{2\theta_i}\right\}\right\}$$

That is,

$$v(2) = \min\left\{w_2(\Im), \min_{1\le i\le m}\left\{\frac{w_2(i)}{2\theta_i} + v(1)\right\}\right\}$$

Similarly, for $j = 3$, we have

$$v_{n+1}(3) = \min_{a \in A}\{w_3(a) + q_3\ (a)v_n(3) + q_{32}\ (a)v_n(2)\}$$
$$= \min\left\{w_3(\Im), \min_{1\le i\le m}\{w_3(i) + (1-3\theta_i)v_n(3) + 3\theta_i v_n(2)\}\right\}$$

and

$$v(3) = \min\{w_3(\Im), \min_{1\le i\le m}\{w_3(i) + (1-3\theta_i)v(3) + 3\theta_i v(2)\}\}$$

Or

$$v(3) - v(2) = \min\{w_3(\Im) - v(2), \min_{1\le i\le m}\{w_3(i) + (1-3\theta_i)(v(3) - v(2))\}\}$$

Therefore

$$v(3) = \min\left\{w_3(\Im), \min_{1\le i\le m}\left\{\frac{w_3(i)}{3\theta_i} + v(2)\right\}\right\}$$

In general we conclude

$$v(j) = \min\left\{w_j(\Im), \min_{1\le i\le m}\left\{\frac{w_j(i)}{j\theta_i} + v(j-1)\right\}\right\} \qquad ; j = 2,3,\ldots,N$$

Q.E.D.

ABOUT THE AUTHORS

Kai-Yuan Cai is a Cheung Kong Scholar (Chair Professor), jointly appointed by the Ministry of Education of China and the Li Ka Shing Foundation of Hong Kong in 1999. He has been a full professor at Beihang University (Beijing University of Aeronautics and Astronautics) since 1995. He was born in April 1965 and entered Beihang University as an undergraduate student in 1980. He received his B.S. degree in 1984, M.S. degree in 1987, and Ph.D. degree in 1991, all from Beihang University. He was a research fellow at the Centre for Software Reliability, City University, London, and a visiting scholar at Purdue University (USA). He was also a Visiting Professorial Fellow with the University of Wollongong, Australia. Dr. Cai has published many research papers in international journals and is the author of three books: *Software Defect and Operational Profile Modeling* (Kluwer, Boston, 1998); *Introduction to Fuzzy Reliability* (Kluwer, Boston, 1996); *Elements of Software Reliability Engineering* (Tsinghua University Press, Beijing, 1995, in Chinese). He serves on the editorial board of the international journal *Fuzzy Sets and Systems*. He also served as guest editor for *Fuzzy Sets and Systems* (1996), the *International Journal of Software Engineering and Knowledge Engineering* (2006), the *Journal of Systems and Software* (2006), and *IEEE Transactions on Reliability* (2011). His main research interests include software reliability and testing, reliable flight control, and software cybernetics.