Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1328–1348
The Journal of Systems and Software
www.elsevier.com/locate/jss

Adaptive software testing with fixed-memory feedback ☆

Kai-Yuan Cai *, Bo Gu [1], Hai Hu, Yong-Chao Li [2]

Department of Automatic Control, Beijing University of Aeronautics and Astronautics, Beijing 100083, China

Received 17 April 2006; received in revised form 31 October 2006; accepted 5 November 2006
Available online 2 February 2007

Abstract

Adaptive software testing is the counterpart of adaptive control in software testing. It means that software testing strategy should be adjusted on-line by using the testing data collected during software testing as our understanding of the software under test is improved. In this paper we propose a new strategy of adaptive software testing in the context of software cybernetics. This new strategy employs fixed-memory feedback for on-line parameter estimations and is intended to circumvent the drawbacks of the assumption that all remaining defects are equally detectable at constant rate and to reduce the underlying computational complexity of on-line parameter estimations. A comprehensive case study with the Space program demonstrates that the new adaptive testing strategy can really work in practice and may noticeably outperform the purely-random testing strategy and the random-partition testing strategy (or collectively, the random testing strategies) in terms of the number of tests used to detect and remove a given number of defects in a single process of software testing and the corresponding standard deviation. In addition, the case study shows that the input domain of the software under test should be partitioned non-evenly for the adaptive testing strategy.

Keywords: Software testing; Adaptive testing; Random testing; Partition testing; Space program; Controlled Markov chain; Adaptive control; Software cybernetics

1. Introduction

1.1. The idea of adaptive testing

Roughly speaking, conventional strategies for software testing including random testing and partition testing are performed in the following steps:

(1) A testing goal or test data adequacy criterion of interest is specified.
(2) A test suite that is expected to achieve the testing goal is generated off-line.
(3) The test cases in the generated test suite are exhaustively or selectively executed.
(4) The testing data are collected on-line during testing and analyzed off-line.
(5) The test suite is updated and the test cases are executed or re-executed if necessary.

In partition testing, the testing goal may be to cover all statements, control flows, data flows or critical paths (Weyuker and Jeng, 1991). The generated test suite often comprises one or more test cases selected from each equivalence class of the input domain of the software under test. All the test cases in the generated test suite are executed. In random or statistical testing, the testing goal may be to perform the given number of tests in accordance with a given probability distribution or stochastic process, and the generated test suite is simply the whole input domain of the software under test or the state space of interest (Myers, 1979; Whittaker and Poore, 1993). The test cases are selectively executed. For mutation testing and regression testing

☆ Supported by the National Natural Science Foundation of China (Grant Nos.: 60633010, 60474006 and 60473067).
* Corresponding author. Tel.: +86 10 82317328; fax: +86 10 82338414.
E-mail address: kycai@buaa.edu.cn (K.-Y. Cai).
[1] Present address: Texas Instruments Inc. at Dallas , 9010 Markville Dr. #209, Dallas, TX 75243, USA.
[2] Present address: IBM China Development Laboratory, 8th Floor Rui-An Square, Huai Hai Road #333, Shanghai 200021, China.

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.11.008

(DeMillo et al., 1978; Rothermel et al., 2001), Step (5) is crucial.

The adaptive software testing strategy proposed in this paper is centered around step (3) mentioned above and is featured with the following characteristics. First, the given testing goal is explicitly defined or formulated. An example testing goal is to minimize the number of tests that cover all control flows. Another example testing goal is to detect as many major software defects as possible with 1000 tests. Second, the software under test is mathematically modeled. Various mathematical models have been available for software such as finite state machines, Petri nets, temporal logics, and process algebraic models (e.g., CSP, CCS). However in this paper the software under test is modeled as a Markov chain under control. Third, the underlying feedback mechanisms in software testing are formulated. In practical software testing, test cases which are selected and executed against the software under test may or may not reveal software failures. Testing results are observed and collected, which may or may not be used to guide the selection of next test cases to be executed. The test case selection process is often feedback guided and should be formulated. Fourth, the testing data are collected and analyzed on-line during testing so that the current status of software testing process and the software under test are continually updated for the purpose of optimal and adaptive control of software testing process. Finally, the testing strategy is synthesized on the basis of control principles, theories and methods. This is in contrast with conventional testing strategies that are mainly defined a priori without a solid theoretical foundation. Overall, the question of concern in the proposed adaptive testing strategy is how test cases in a vast test suite or the whole input domain of the software under test are selectively executed in an optimal manner.

Fig. 1.1 shows the diagram of adaptive software testing that comprises four major components, with the mathematical notation being explained in Section 2. The rectangular containing the software under test denotes the first component, which serves as the controlled object and is modeled by a controlled Markov chain (see Section 2). The rectangular containing the database denotes the second component, which collects the testing results on-line during testing. The rectangular containing the testing strategy denotes the third component, which serves as the controller and is synthesized on the basis of the given testing goal and the theory of controlled Markov chains. It receives the history of testing data and the outputs of the parameter estimation component as its inputs and decides which test case should be selected and executed next. The rectangular containing the parameter estimation scheme denotes the fourth component, which estimates the parameters of concern (e.g., the initial number of software defects, the defect detection rates by distinct test cases) by using the testing data collected on-line. These four components constitute two distinct feedback loops. The first (inner) feedback loop consists of the testing strategy, the software under test, and the database. The next test cases are selected and executed by using the history of testing data under the assumption that the testing strategy or the corresponding parameters such as the defect detection rates keep invariant during testing. The second (outer) feedback loop consists of the testing strategy, the software under test, the database, and the parameter estimation scheme. The main functionality of the outer feedback loop is that the history of testing data is used to estimate the required parameters of the software under test on-line and the testing strategy is updated or improved accordingly. The improvement may lead the random testing to switching from one test distribution (e.g., uniform distribution) to another test distribution (e.g., non-uniform distribution). It may lead partition testing from one partitioning of the input domain of the software under test to another partitioning. It may also lead data flow testing to boundary value testing. This is just to simulate the scenario that 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. In this paper the improvement leads to better test case selection schemes as a result of better estimates of defect detection rates. Without the outer feedback loop, the corresponding testing strategy reduces to a usual feedback guided or non-adaptive controller (testing strategy), which specifies what test suite or what next test cases should be generated or selected. However an adaptive software testing strategy specifies what next testing policy should be employed and thus in turn what test suite or next test cases should be generated or selected in accordance with the new testing policy. Adaptive testing specified in Fig. 1.1 is the adaptive control counterpart in software testing.

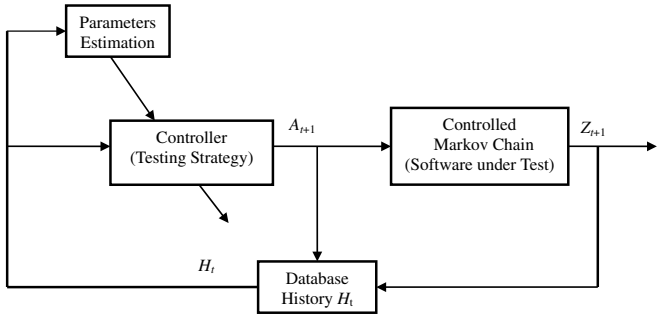

Fig. 1.1. Diagram of adaptive software testing.

1.2. Research questions

The idea of adaptive software testing was first proposed in our previous work (Cai, 2002; Cai et al., 2005), where the whole history of testing data was used in both the feedback loops and simulation results were presented to justify the advantages of adaptive testing over random testing. However several research questions have not been well addressed.

(1) How to improve the unrealistic assumptions of software testing processes. A major assumption adopted

in our previous work (Cai, 2002; Cai et al., 2005) is that all remaining software defects are equally detectable in the whole process of software testing and the contribution of each single remaining defect to software failure rate keeps constant. More often than not, this assumption is unrealistic and should be replaced or compensated to some extent. This is the main motivation for the work presented in this paper.

(2) How to avoid aging testing data in on-line parameter estimation and reduce the on-line computational burden. In our previous work (Cai, 2002; Cai et al., 2005), the whole history of testing data was used in both the feedback loops. This causes two major problems. First, testing data may age in the sense that testing data collected in the early stage of software testing may not be highly relevant to the current status of the software under test since the software under test may have dramatically been changed as a result of defect removals during testing. It does not seem most appropriate to use the whole history of testing data for parameter estimations. Second, using the whole history of testing data to do on-line parameter estimations may induce a severe computational complexity problem as software testing proceeds. For large scale software systems the whole history of testing data may contain thousands of tests. Then how to adopt partial history of software testing in the feedback loops?

(3) Is the adaptive software testing strategy effective for real software programs. Only simulation results are presented in our previous work (Cai, 2002; Cai et al., 2005), which are certainly valuable for demonstrating how adaptive software testing may behave. It is common practice in control engineering that simulation results are employed to justify the behavior of control systems under consideration. However simulation results may not be enough for demonstrating the effectiveness of adaptive software testing. This is because the software under test as well as the corresponding software testing process is complex. Modeling of the software under test and the software testing process is subject to abstractions and unrealistic assumptions. This can possibly lead to big gaps between simulation results and practical behavior of software testing. Then do abstractions and unrealistic assumptions render invalid conclusions of adaptive software testing? Obviously, this should be answered by applying the adaptive testing strategy to real software programs. Suppose testing results with real software programs demonstrate that the adaptive software testing strategy can noticeably outperform existing software testing strategies such as random testing strategies. It is then reasonable to say that adaptive software testing strategy is really effective, though some of unrealistic assumptions are adopted for synthesizing the adaptive software testing strategy.

(4) How to partition test suites for adaptive software testing. Although the proposed adaptive testing strategy is mainly concerned with how to select test cases from the given test suite, various steps in conventional software testing become highly interleaved. Then how will other steps affect the effectiveness of the proposed adaptive testing strategy. In particular, how to partition the given test suite for the purpose of adaptive testing?

In this paper we introduce the idea of fixed-memory feedback for adaptive software testing. More specifically, the whole history of testing data is used in the inner feedback loop for the sake of mathematical tractability so that a closed-form testing strategy or controller can be derived on the basis of the theory of controlled Markov chains. However, only fixed amount of latest testing data is used in the outer feedback loop for the parameter estimation in an attempt to serve several purposes. First, aging testing data are partially avoided in the parameter estimation. Second, the on-line computational complexity problem is alleviated to some extent. Finally, the unrealistic assumption that all remaining software defects are equally detectable in the whole process of software testing and the contribution of each single remaining defect to software failure rate keeps constant is partially compensated. We then apply the resulting adaptive testing strategy with fixed-memory feedback to a real software program, the Space program. This gives us a chance to validate or invalidate the effectiveness of adaptive software testing. The experimental results are more convincing than the mathematical assumptions. If the experimental results show that the proposed adaptive testing strategy outperforms existing software testing strategies, then possibly unrealistic assumptions imply that the proposed adaptive testing strategy is effective and can be further improved by adopting more realistic assumptions. If the experimental results are not in favor of the proposed adaptive software testing strategy, then the effectiveness of adaptive software testing is highly questionable, no matter however reasonable the underlying assumptions may looks.

1.3. Related works

Feedback is ubiquitous in software systems and software processes. It is not new to acknowledge the possible role of feedback in software engineering in general, and in software testing in particular. Malaiya proposes an anti-random testing strategy in comparison with the random testing strategy. In anti-random testing a test case is not selected at random. Rather, a test case is selected such that its total distance from all previously test cases is maximized (Malaiya, 1995; Yin et al., 1997). The so-called adaptive random testing proposed by Chen et al. (2001) shares a similar idea with anti-random testing, where failure patterns are used as a source of feedback information. In regression testing feedback may play positive or negative role in test case prioritization (Elbaum et al., 2002). A

similar problem can be observed in unit testing (Xie and Notkin, 2003; Pacheco and Ernst, 2005): Given a program unit u, a set S of existing tests for u, and a test t from a set S' of unselected tests for u, does the execution of t exercise at least one new feature that is not exercised by the execution of any test in S? In the on-the-fly, model-based testing of reactive systems (Nachmanson et al., 2004; Veanes et al., 2005; Campbell et al., 2005), software testing is treated as a game playing problem and test cases are generated on-the-fly by using testing feedback information during the process of software testing. The test derivation from a model program and test execution is combined into a single algorithm in the testing tool SpecExplorer developed by Microsoft Research. However there are two differences between the works of references (Nachmanson et al., 2004; Veanes et al., 2005; Campbell et al., 2005) and the work presented in this paper. First, references (Nachmanson et al., 2004; Veanes et al., 2005; Campbell et al., 2005) follow an implicit paradigm: generate and execute test cases one by one till a target state is reached. The implicit paradigm followed by the work presented in this paper is different: collect a test suite of a huge size (which may be legacy test suites or even the whole input domain of the software under test) and then select "optimal" test cases from the test suite. Second, the underlying control theory that guides the on-the-fly testing or the synthesis of the required controller (testing strategy) is not explicitly formulated, whereas the work presented in this paper is based on the theory of controlled Markov chains.

The idea of formulating the feedback mechanisms in software processes in a rigorous manner and treating software testing as a control problem is rather new. The work of Cangussu et al. (2001, 2002) is one of examples in this line, in which a deterministic continuous-time state-space model was developed for testing process and linear control systems theory was adopted to guide software testing management. Another example is the adaptive software testing strategy that was first proposed in the framework of the controlled Markov chains (CMC) approach to software testing (Cai, 2002; Cai et al., 2005). The CMC approach distinguishes itself from related works in several aspects. First, software testing is treated as a control problem and the underlying feedback mechanism is formalized, quantified and optimized. Second, the software under test is modeled as a controlled Markov chain which is characterized as discrete-time discrete-event process and the theory of controlled Markov chains is used to synthesize the required testing strategy. Third, adaptive software testing is proposed and defined as the counterpart to adaptive control in software testing. Two feedback loops are adopted as shown in Fig. 1.1. This dramatically contrasts with related works that adopt only a single feedback loop. Our previous work (Cai, 2002; Cai et al., 2005) was the first one which treated software testing as an optimal and adaptive control problem and put the control problem into a stochastic framework. However a drawback of our previous work was that only simulation results were presented.

1.4. Organization of the paper

The rest of the paper is organized as follows. Section 2 formulates the new adaptive software testing strategy. Section 3 describes the subject program and the testing algorithms used in our case study. The case study comprises two experiments. The experimental results are presented in Sections 4 and 5. In Section 6 we analyze the experimental results. Concluding remarks are contained in Section 7.

2. New adaptive software testing strategy

In response to the first and second research questions identified in Section 1, in this section we show how to adopt partial history of testing data in adaptive testing. We formulate the four components in Fig. 1.1. Let

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

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

In this paper an action refers to selection and execution of a test case from the input domain of the software under test. However action can be interpreted more widely. It can refer to a software run, or selection of an equivalence class of test cases for a specific testing technique. It can also refer to selection of a software testing technique such as boundary testing, path testing or random testing. An action can even mean selection of a tester. Different actions have different defect detection ability, or we can say that the software demonstrates different testability under different actions. A control policy (testing strategy) specifies which action should be taken at every time.[3] Since actions are taken one by one, the corresponding behavior of software input–output is modeled in the discrete-time domain.[4] Further, software state is defined in terms of Y_t. The software is said to be in state j at time t if $Y_t = j$.

A simple model of software testing process is based on the following assumptions (Cai, 2002).

(1) The software contains N defects at the beginning ($t = 0$).
(2) An action taken at one time detects at most one defect.

[3] In existing testing literature, the terms testing process and testing strategy are mainly concerned with software management involving people and organizations. In this paper these terms are borrowed to refer to the technical aspects of software testing while software testing is treated as control problem. This is for two reasons. First, if these terms were not borrowed, then new terms should be invented and this might lead to unnecessary confusion. Second, although the CMC approach is mainly applied to the technical aspects in this paper, this by no means implies that it cannot be applied to formulate the management aspects of software testing.

[4] In software reliability modeling, the time domain can be continuous or discrete (Cai, 2000).

(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

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

where θ_i can be interpreted as the probability of each defect being detected by action i.
(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.

We are interested in how many actions or tests are required to remove all the N defects from the software under test. That is, we intend to make the software under test defect-free. Suppose that the first $\tau - 1$ actions (at times $0, 1, \ldots, \tau - 2$) detect (and remove) $N - 1$ defects and the τth action (at time $\tau - 1$) detects the last defect. Then $Y_{\tau-1} = 1, Y_\tau = 0$, and τ is referred to as the first-passage time to state 0 ($Y_\tau = 0$). That is, τ denotes the number of actions that are performed to remove all the N defects. The software under test should move from the initial state $Y_0 = N$ to the target state $Y_\tau = 0$ during testing. Further let ω denote the testing strategy that is adopted in the process of software testing. ω specifies how test cases are selected one by one on-line during software testing. In conventional software testing it may refer to a partition testing strategy or a random testing strategy. Here ω refers to the adaptive testing strategy that we want to derive on the basis of the theory of controlled Markov chains. Note that ω is not specified a priori. It can be treated as a variable whose value should be determined with respect to a given testing goal (e.g., Eq. (2.1) below). Obviously, different ω leads to different value of τ. For example, consider that a software program with five remaining defects is subjected to random testing with a test suite of 1000 test cases. The testing strategy, ω_1, which selects test cases in accordance with the uniform probability distribution such that each test case has a probability of 1/1000 being selected at every time, may perform 100 tests to detect and remove the five remaining defects. However the testing strategy, ω_2, which only selects from a given class of 200 test cases such that each of the 200 test cases has a probability of 1/200 being selected at every time, may only performs 70 tests to detect and remove the five remaining defects. Even for a given ω, due to the uncertainty associated with the software testing process and the software under test, the value of τ cannot be determined a priori. There is no guarantee that ω_1 always requires 100 tests to detect and remove the five remaining defects. It may also happen to require 98 or 112 tests. τ should better be treated as a random variable. Consequently, the total cost incurred in a testing process is a random variable too. The expected total cost (from the first action at time 0 to the last action at time $\tau - 1$) is denoted as

$$J_\omega(N) = E_\omega \sum_{t=0}^{\tau-1} W_{Y_t}(A_t) \tag{2.1}$$

Note that, $W_{Y_t}(A_t)$ is the cost incurred by action A_t taken at time t (from Assumption (8) above), and τ is the total number of actions required to detect and remove all the N defects from the software under test, $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ is thus the total cost incurred in a single testing process that detects and removes all the N defects. $E_\omega \sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ represents the mathematical expectation of $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ under testing strategy ω. In order to simplify the mathematical notation, this quantity is denoted $J_\omega(N)$, as shown in Eq. (2.1). A natural objective is to determine the testing strategy ω that minimizes $J_\omega(N)$. Such a testing strategy removes all the N defects at the least expected total cost. A special case is that there holds

$$w_j(i) \equiv 1 \quad \forall j \neq 0, \ \forall i$$

Then the corresponding optimal testing strategy detects and removes all the N defects at least expected number of actions (tests), $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t) = \tau$.

Remarks

(1) Assumptions (1)–(7) define a model for the software under test or *the first component of* Fig. 1.1. This model describes how many distinct inputs (actions) acceptable to the software and how the software can possibly behave under the distinct inputs. On the other hand, Assumptions (8) and (9) and Eq. (2.1) specify the testing goal of interest. The assumption $W_{Y_t}(A_t = i) = 0$ for $Y_t = 0$ is adopted only for mathematical convenience. In practice, no further actions will be taken in the state of $Y_t = 0$ which is the target state according to Assumption (5).
(2) Assumption (3) implies that a defect is removed immediately upon being detected. This is unrealistic for software system testing. A more realistic scenario is that debugging is carried out after a number of defects is detected. This will be discussed in the future work of adaptive testing.
(3) Assumption (5) and Eq. (2.1) imply that the initial number of remaining defects is known. This can hardly be true. A more realistic testing goal is to

detect and remove as many defects as possible with a given number of tests. This problem is tackled in Cai et al. (2005) and different from the problem tackled in Cai (2002).

(4) Assumption (7) indirectly implies that software defects are equally detectable. This assumption is not realistic and is the main motivation for us to propose a new adaptive testing strategy in this paper.

(5) More irrational can be observed from the above assumptions. For example, Assumption (2) is not true in software testing practice. In early stage of software testing, execution of a test case may detect more than one defect. If an action refers to selection of a software testing technique such as boundary testing, path testing or random testing, then it is likely for one action to detect more than one defect. However, as a preliminary study for a new line that puts software testing into the framework of control theory principles, this paper is not intended to argue for rationale of various assumptions. Rather, it presents experimental results that demonstrate that the adopted assumptions on the whole are acceptable. In turn, it is feasible to formulate the underlying feedback mechanisms in software testing, and to quantify and optimize the software testing process. The irrational of specific assumptions implies the proposed testing strategies based on control theory principles can be improved to achieve better results. Modeling looks indispensable for systematic researches on software testing, *which must incur abstractions*, *assumptions*, *and irrational*.

(6) It is important to understand the *role* of the above assumptions. The assumptions are taken for deriving the desired testing strategy or controller. Better assumptions may lead to better testing strategy. However this by no means indicates that the resulting testing strategy based on simplifying or even unrealistic assumptions cannot be adopted. Further, the software under test is not required to strictly fit the assumptions. Large deviations from these assumptions only mean that the resulting testing strategy may perform poorly for the software under test. Actually, similar scenarios can extensively be observed in control engineering. Real-world systems can seldom be linear. However linear system models are widely employed to synthesize the required linear or nonlinear controllers for the real-world systems. If the resulting controllers based on the (simplifying or unrealistic) linear system models can work well for the real-world systems that may be highly nonlinear, then we say that the controllers are robust. Therefore, if the resulting testing strategy based on the above (unrealistic) assumptions can behave well in software testing experiments for the software under test which are unlikely to fit the above assumptions, then we say that the testing strategy is robust. Without this understanding, one might argue that the resulting testing strategy were useless and should not be trusted. The experimental results were then misinterpreted.

The above assumptions and Eq. (2.1) define a simple controlled Markov chain as shown in Fig. 2.1. Denote $z_{-1} = 0$. Let $\{z_t, t = 0, 1, 2, \ldots\}$ be realization of $\{Z_t, t = 0, 1, 2, \ldots\}$. Further let $\{a_t, t = 0, 1, 2, \ldots\}$ be realization of $\{A_t, t = 0, 1, 2, \ldots\}$. That is, at time t, action a_t is taken which generates an observation of $Z_t = z_t$. Then $(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t) = H_t$ represents the whole history of testing data and constitutes *the second component of* Fig. 1.1. By using H_t, we are able to derive the optimal controller or *the third component of* Fig. 1.1 as demonstrated in our previous work (Cai, 2002).

More specifically, for a function $f(x)$ defined on a given domain D, we denote $\arg\min_x f(x)$ as the value of x that minimizes $f(x)$ over D. Fox example, $f(x) = x^2$ and $D = [0.5, 1]$. Then $\arg\min_x f(x) = 0.5$. Further let

$$v(1) = \min_{1 \leqslant i \leqslant m} \left\{ \frac{w_1(i)}{\theta_i} \right\} \tag{2.2}$$

$$v(j) = \min_{1 \leqslant i \leqslant m} \left\{ \frac{w_j(i)}{j\theta_i} + v(j-1) \right\}; \quad j = 2, 3, \ldots, N \tag{2.3}$$

Recall that $w_1(i)$ is the cost of action i taken while the software under test stays in state 1 (contains only one defect), and θ_i is the probability that a defect is detected by action i. Thus $\frac{w_1(i)}{\theta_i}$ is the average cost that is required to detect the defect if action i is continually applied, and $v(1)$ denoted in Eq. (2.2) is the least cost required to detect the defect if an identical action is continually applied. Similarly, $\frac{w_j(i)}{j\theta_i}$ is the average cost that is required to detect a defect while the software under test contains j defects if action i is continually applied, and $v(j)$ denoted in Eq. (2.3) is the least cost required to detect all the j defects.

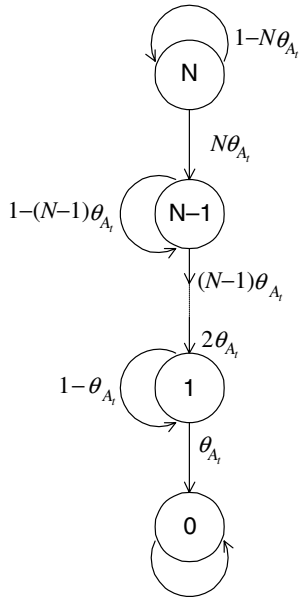

Fig. 2.1. Software state transition under test.

For the testing goal specified in Eq. (2.1), in our previous work (Cai, 2002) we have shown that the optimal action that should be taken is $\arg v(1) = \arg\min_{1\leqslant i\leqslant m}\left\{\frac{w_1(i)}{\theta_i}\right\}$ in state $j=1$ and is $\arg v(j) = \arg\min_{1\leqslant i\leqslant m}\left\{\frac{w_j(i)}{j\theta_i} + v(j-1)\right\}$ in state $j \geqslant 2$. 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. Note this optimal control policy is Markovian in the sense that the optimal action that should be taken in a state depends on the current state and is independent of the history of software states. Overall, $\arg v(i)$ specifies the optimal controller or testing strategy that achieves the given testing goal.

Now let us formulate *the four components of* Fig. 1.1. Note that the unknown parameters $\theta_1, \theta_2, \ldots, \theta_m$ are involved in Eqs. (2.2) and (2.3). In order to implement the optimal testing strategy (the third component of Fig. 1.1), these unknown parameters must be estimated on-line. This leads to an adaptive software testing strategy (the optimal testing strategy with the on-line parameter estimation scheme) as shown in Fig. 1.1. Each time the estimates of the unknown parameters are updated by using the history of testing data and treated as the true values of these parameters in decision-making. This is the so-called the certainty-equivalence principle in adaptive control (Hernandez-Lerma, 1989). Now we specify the parameter estimation scheme in Fig. 1.1 that explains how the parameters of concern are estimated and updated on-line.

Let

$$r_t = \left(N - \sum_{j=-1}^{t-1} z_j\right)\theta_{a_t} \tag{2.4}$$

r_t represents the probability that action a_t detects a defect and is just the expected value of Z_j. Given $z_0, z_1, \ldots, z_t$ and $a_0, a_1, \ldots, a_t$, we can use the least squares method to do parameter estimation. Let

$$L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t) = \sum_{j=0}^{t}(z_j - r_j)^2 \tag{2.5}$$

The $m+1$ parameters $N, \theta_1, \theta_2, \ldots, \theta_m$ are estimated by minimizing the function $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$, and the resulting estimates, denoted as $N^{(t+1)}, \theta_1^{(t+1)}, \theta_2^{(t+1)}, \ldots, \theta_m^{(t+1)}$, are function of the whole history of testing data up to the current instant of time,

$$\begin{aligned} N^{(t+1)} &= N(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t) \\ \theta_i^{(t+1)} &= \theta_i(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t); \quad i = 1, 2, \ldots, m \end{aligned} \tag{2.6}$$

A commonly used way to obtain these estimates is to differentiate $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ with respect to each parameter of concern and obtain a system of nonlinear equations.[5] However $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ is not always differentiable. Even if it is differentiable, sometimes the resulting system of nonlinear equations may not have a solution. A more general way of doing parameter estimation is to minimize $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ directly by using genetic algorithms. However for large scale software systems the number of actions (tests) that should be applied in total is huge.[6] Using the whole history of testing data to do on-line parameter estimation induces a severe problem of computational complexity.

On the other hand, we may doubt if it is necessary to use the whole history of testing data to do parameter estimation. Actually, a major threat to the effectiveness of the proposed adaptive testing strategy should be Assumption (7). This assumption implies that each of remaining software defects contributes equally to software failure rate. It also implies that the contribution of a remaining software defect keeps constant during the whole software testing process till the defect is detected and removed. From the viewpoint of parameter estimation, using whole history of testing data implies that the estimates of the $m+1$ parameters should each eventually converge to a constant. All these implications make Assumption (7) questionable. Although assumptions of this kind are common in software reliability modeling (Cai, 1998; Xie, 1991; Lyu, 1996), it is generally agreed in software engineering community that different software defects have different probabilities of being detected and these probabilities may be changed during software testing since the nature or properties of the software under test may be changed as the remaining software defects are detected and removed. Software defects may not be independent and there are coupling effects among them. The testing data collected in early stage of software testing may age and thus fail to provide useful guides to test case selections in the late stage of testing. Further, new software defects may be introduced during the process of software defect removals and this makes Assumption (3) invalid. The parameter N should better be treated as time-varying parameter.

Therefore we should have a better way of modeling software testing process.[7] A possible way is to develop a set of more realistic assumptions in place of Assumptions (1)–(9). For example, we may assume that the software failure rate is a time-varying nonlinear function of the number of

[5] This paper is not intended to argue for which parameter estimation method is best. Methods other than genetic algorithms may also be adopted.

[6] In the experiments presented in Sections 4 and 5, the total number of tests applied in a single testing process may be up to several thousands.

[7] This by no means implies that the simple controlled Markov chain model of adaptive software testing proposed in our previous work (Cai, 2002) could not work. The major underlying reason is that the adaptive testing strategy follows a feedback philosophy and the history of testing data is used to optimize software testing process on-line. This paper is aimed to improve, and is not aimed to invalidate our previous simple model of software testing process. Assumption (7) actually means that the software failure rate is a linear function of the number of remaining software defects. Alternatively, we may assign a nonlinear functional structure for the software failure rate. Whether a nonlinear software failure rate function helps the adaptive testing strategy will be investigated in the future.

remaining software defects. We may also assume that the number of defects detected by an action taken at one time follows a probability distribution. This may lead to a difficult synthesis problem for the required optimal testing strategy since the underlying controlled Markov chains may be complicated. Additional parameters for the probability distribution of the number of number of defects detected by an action must be estimated. In this paper we follow an alternative way. We keep Assumptions (1)–(9) and thus Eqs. (2.2) and (2.3) are valid. As shown in our previous work (Cai, 2002), Eqs. (2.2) and (2.3) are obtained in theory by assuming the whole history of testing data is available for feedback in the first feedback loop of Fig. 1.1. They determine an optimal testing strategy. However, the optimal testing strategy is Markovian and actually we do not need the whole history of software states or testing data to make an optimal decision on-line if the required software parameters are known a priori. Further, in this paper we avoid using the whole history of testing data to do parameter estimations. We only use latest testing data of fixed memory. This can be intuitively justified by the following observation. Although Assumptions (3) and (7) should not be valid in the whole process of software testing, they may hold and make sense in a short period of software testing. If we only use latest testing data of fixed memory to do parameter estimations, then the $m+1$ parameters $N, \theta_1, \ldots, \theta_m$ no longer behave as a constant in the whole history of software testing. They are intrinsically time-varying. Consequently, we can argue that we do not strictly adhere to Assumptions (3) and (7) in software testing. The irrational of these assumptions is compensated in an indirect manner. We only assume that at time t the j software defects collectively contribute $j\theta_t$ to the software failure rate which is a function of time.

Specifically, we use the testing data generated by the latest l actions (tests) to do parameter estimations. Let

$$L(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t) = \sum_{j=t_0}^{t} (z_j - r_j)^2 \tag{2.7}$$

where

$$t_0 = \min\{0, t - l + 1\} \tag{2.8}$$

Using the genetic algorithm implemented in the MATLAB to minimize the objective function $L(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t)$, we can obtain estimates for the $m+1$ parameters $N, \theta_1, \theta_2, \ldots, \theta_m$ as follows:

$$\begin{aligned} N^{(t+1)} &= N(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t) \\ \theta_i^{(t+1)} &= \theta_i(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t); i = 1, 2, \ldots, m \end{aligned} \tag{2.9}$$

Following the certainty-equivalence principle and using Eqs. (2.7)–(2.9) to replace Eqs. (2.5) and (2.6), we arrive at the following adaptive testing strategy:

Step 1: Initialize parameters. Set

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

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

A_0 denotes the first action that is performed for software testing.

Step 2: Set l to a given value. Let $t_0 = \min\{0, t - l + 1\}$. The testing data collected at times $\{t_0, t_0 + 1, \ldots, t\}$ are used to update the estimates of the parameters at time $t + 1$ in Step 3.

Step 3: Estimate parameters by minimizing $L(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t)$ (Eq. (2.7)) and obtain

$$\begin{aligned} N^{(t+1)} &= N(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t) \\ \theta_i^{(t+1)} &= \theta_i(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t); \quad i = 1, 2, \ldots, m \end{aligned}$$

If θ_i doesn't appear in $L(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t)$, or action i was not taken in the latest l actions, then we pick up a value from the unity interval $[0, 1]$ at random in accordance with the uniform distribution and assign the value to $\theta_i^{(t+1)}$. (This step specifies the parameter estimation scheme.)

Step 4: Decide the optimal action.[8]

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

(This step specifies the optimal controller or testing strategy.)

Step 5: Observe the testing result $Z_{t+1} = z_{t+1}$ activated by the action A_{t+1}.

Step 6: Update the current software state, which represents the number of remaining defects

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

Step 7: Set $t = t + 1$.

Step 8: If a given testing stopping criterion is satisfied, then stop testing[9]; otherwise go to Step 3.

[8] The experience with the experiments presented in Sections 4 and 5 showed that the adaptive testing strategy took more time to select test cases than the random testing strategy indeed. Roughly, the random testing strategy could select and apply 10 test cases in a single second of time, whereas this number reduced to 3 or 5 for the adaptive testing strategy. However this also implied the time complexity should not be a hurdle for the adaptive testing strategy to be applied in practice.

[9] 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. For example, a more practical stopping criterion is that Y_{t+1} is below some given threshold. In the experiments described in Sections 6 and 7, software testing was stopped till a given number of defects is removed since the number of seeded defects in the software under test is known a priori.

Up to this point we can see that all the four components in Fig. 1.1 are specified. Only fixed-memory of feedback is used for on-line parameter estimation, although the whole history of testing data is adopted to derive the optimal testing strategy for the sake of mathematical tractability in theory. The irrational of the mathematical model (assumptions) for the software under test is partially compensated by the parameter estimation scheme in an indirect way. Software testing is treated as an optimal and adaptive control problem in a reasonable manner.

One may still have doubts about the rationale of the proposed adaptive testing strategy. In particular, how to choose the size of the memory is not specified in a rigorous manner.[10] The adaptation scheme (in the outer loop of Fig. 1.1 that changes the testing strategy from one method to another by updating the estimates of the parameters) is not guaranteed to behave in an optimal manner and thus may impair the effectiveness of defect detection. Since different defects have different probabilities of being detected by different testing methods, a method that works well for the detection of the first defects may turn out as completely inefficient in the last stage of testing when only those defects have remained in the software that are immune against just this method. The adaptive testing strategy may manage the necessary change to another method after a first method has become inefficient, using the effect that in this situation, the estimates of the defect detection rates become gradually smaller and smaller. However it is not clear how the required change of methods occurs at the optimal point in time. Fortunately, although it is not feasible to reject them in theory for the time being, all these doubts should be alleviated to a large extent by the experimental results presented in the rest of this paper. These doubts actually define several topics that deserve further investigation.

3. Testing set-up

In order to address the third research question identified in Section 1, that is, whether the adaptive testing strategy is effective, we applied the adaptive testing strategy, the random testing strategy and the partition testing strategy to a subject program. If the adaptive testing strategy uses fewer tests to detect more defects than the random testing strategy and the partition testing strategy, then it is reasonable to say the adaptive testing strategy makes meaningful contribution to software testing.

3.1. Subject program

The subject program is called Space program and was already used in the study of random testing (Vokolos and Frankl, 1998) and that of regression testing (Rothermel et al., 2001).[11] Rothermel et al. described it as follows (Rothermel et al., 2001):

"Space consist of 9564 lines of C code (6218 executable), and functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, Space outputs an array data file containing a list of array elements, positions, and excitations; otherwise, the program outputs error message".

In previous studies (Rothermel et al., 2001; Vokolos and Frankl, 1998), 38 mutants were generated from the Space program, each containing one distinct defect. That is, there were 38 distinct defects in total. For the purpose of validating the proposed adaptive testing strategy, we injected all the 38 distinct defects into the Space program to generate a single faulty version which was then subjected to testing. The test suite (or the input domain) used in our case study comprises 13,466 test cases, each being an ADL file. These test cases were exactly those used in the previous studies (Rothermel et al., 2001; Vokolos and Frankl, 1998). The original Space program without injected defects served as the test oracle. That is, a test case was applied to both the original Space program and the Space program with injected defects. Any discrepancy between the corresponding outputs meant that a failure was observed and at least one injected defect was detected. Among the 38 distinct defects, two (defects No. 34 and No. 38) were non-detectable by any test case of the test suite.

3.2. Testing process

Since the test suite contained more than ten thousands test cases, we partitioned these test cases into several disjoint classes (equivalence classes or subdomains).[12] A testing strategy selected a test case in two sub-steps. First, an equivalence class was selected in accordance with the testing strategy. Second, a test case was selected from the equivalence class at random in accordance with a uniform probability distribution (depending on the size of the equivalence class). The test suite kept invariant throughout the various software testing processes.

Different testing strategies followed different testing processes or testing algorithms. For the adaptive testing strategy proposed in Section 2, we adopted the algorithm (Algorithm A) as described below. The purpose of this algorithm was to inject all the 38 defects into the Space program,

[10] It is an open problem in theory how to determine the best size of the memory. However our recent experimental studies indicate that, suppose the whole testing process can be divided into three distinct stages in the terms of the growth rate of the cumulative number of detected defects, the size of the memory should be roughly equal to the length of the middle stage of software testing.

[11] The Space program, the injected defects, and the test suite were obtained from Gregg Rothermel.

[12] We should note that the adaptive testing strategy proposed in Section 2 does not require the subdomains are disjoint.

and then tested the Space program with injected defects to detect and remove all the injected defects except the two non-detectable defects. This algorithm defined a single process or a trial of software testing and this single process was repeated for many times in our experiments to evaluate the effectiveness of the adaptive testing strategy from a statistical viewpoint. A test environment, TestE, was built up to automate the test case selections and software testing.

Algorithm A

BEGIN

Step 1 Prepare the faulty version of the Space program that contain 38 distinct defects and the original versions of Space program that serves as the testing oracle.
Step 2 Assign an value to l, the number of latest tests that are used to do on-line parameter estimation.
Step 3 Assign the number of equivalence classes that the given test suite is partitioned into.
Step 4 Apply the adaptive testing strategy defined in Section 2 to select an equivalence class from the given test suite.
Step 5 Select a test case at random from the equivalence class, which was identified in Step 4, in accordance with a uniform probability distribution (depending on the size of the equivalence class).
Step 6 Apply the selected test case to the faulty and original versions of the Space program.
Step 7 If a failure is observed, then a failure-causing defect is removed from the faulty version of the Space program.
Step 8 If only the two non-detectable defects (defects No. 34 and No. 38) are remaining in the faulty version of the Space program, stop the testing process.
Step 9 Go to Step 4.

END

For random testing, we adopted the following testing algorithm.

Algorithm R

BEGIN

Step 1 Prepare the faulty version of the Space program that contain 38 distinct defects and the original versions of Space program that serves as the testing oracle.
Step 2 Assign the number of equivalence classes that the given test suite is partitioned into.
Step 3 Pick-out an equivalence class at random in accordance with a uniform probability distribution (depending on the given number of equivalence classes).
Step 4 Select a test case at random from the equivalence class, which was picked-up in Step 3, in accordance with a uniform probability distribution (depending on the size of the equivalence class).
Step 5 Apply the selected test case to the faulty and original versions of the Space program.
Step 6 If a failure is observed, then a failure-causing defect is removed from the faulty version of the Space program.
Step 7 If only the two non-detectable defects (defects No. 34 and No. 38) are remaining in the faulty version of the Space program, stop the testing process.
Step 8 Go to Step 4.

END

Remarks

(1) For testing Algorithm A, Steps 1–3 were performed manually to initialize the test environment TestE, and the other steps were performed by TestE automatically. Steps 1–9 removed all the 36 detectable defects injected into the Space program and defined a single process or trial of software testing.
(2) Two parameters should be initialized before Algorithm A can be applied. One is l, which denotes the amount of feedback information. The other is the number of equivalence classes which the test suite is partitioned into.
(3) For testing Algorithm R, the parameter l was not required since no feedback information was adopted for test case selections. Rather, a test case was selected at random in two steps as described in Steps 3 and 4 of Algorithm R.
(4) In order to stick to Assumption (2) of Section 2, in both testing algorithms one and only failure-causing defect is removed upon a failure is revealed. If more than one defect is detected in an action (by applying a single test case), one of these detected defects is randomly selected and removed.
(5) Two defects are non-detectable in the sense that none of the test cases in the given test suite is able to detect them. This does not mean that no other test cases outside the given test suite can detect them.
(6) Since Assumption (7) indirectly implies that software defects are equally detectable, one may doubt if it was reasonable to inject all the 38 distinct defects to make a single faulty version of the Space program. Distinct defects might be correlated or masked each other. To justify the rational of adopting a single faulty version, we have the following observations. First, as pointed out in Remark 6 of Section 2, the role of the model assumptions should not be misunderstood. The robustness of the adaptive testing strategy tolerated the gap between Assumption (7) and the subject program to a certain extent. Second, the same subject program was

adopted for both the adaptive testing strategy and the random testing strategy, and this offered a common basis for comparing the performance of the two strategies. Finally, among the 38 distinct defects, only three defects could be masked by other defects, that is, the existence of other particular defects could make the three defects non-detectable. It is fair to say the masking effects between the 38 defects were marginal ($3/38 \times 100\% = 7.89\%$).

(7) Here we should note that the random testing strategy defined by Algorithm R is slightly different from the so-called random testing or partition testing in the existing literature (Weyuker and Jeng, 1991; Duran and Ntafos, 1984). The existing random testing strategy follows a probability distribution to select test cases from the test suite or input domain of the software under test and no partitioning is made into the input domain. The testing strategy defined by Algorithm R with the number of equivalence class being 1 is actually the purely-random testing strategy since a uniform probability distribution is used to select test cases from the input domain. The existing partition testing strategy selects an equivalence class in accordance with some criterion and then picks up one or more test case from the selected equivalence class at random. How many test cases are selected from an equivalence class depends on application and is often an art. The testing strategy defined by Algorithm R is a generalized 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 on-line according to Algorithm R. Note that in existing partition testing the number of test cases selected from each equivalence class is given a priori and can be viewed as a special form of random variables. Thus the testing strategy defined by Algorithm R can more accurately be referred to as 'random-partition' testing strategy. However we should also note that the random-partition testing strategy is also a form of the existing random testing strategy. If all the equivalence classes of the input domain are disjoint and each contain an equal number of distinct test cases, then according to Algorithm R, every distinct test case has an equal probability of being selected. This is just the purely-random testing strategy. If the equivalence classes overlap or contain different number of test cases, then different test cases may have different probabilities of being selected and this determines a random testing strategy with a non-uniform probability distribution. We will revisit this in Section 6.1.

(8) Ideally, a conventional partition testing strategy should be applied. That is, the required test cases are first generated according to a white-box criterion or a functional criterion and then exhaustively executed. These test cases constitute a number of equivalence classes in the sense of the white-box criterion or the functional criterion. However the generated test suite may not be the one that was adopted in the previous empirical studies (Rothermel et al., 2001; Vokolos and Frankl, 1998) and one may doubt if the adopted white-box criterion or the functional criterion is reasonable. It is known that the strength of partition testing heavily depends on a suitable definition of the adopted white-box criterion or the functional criterion. In order to alleviate this doubt, in the experiments presented in this paper the test suite adopted in the previous empirical studies (Rothermel et al., 2001; Vokolos and Frankl, 1998) was partitioned into various "equivalence classes" at random. This implicitly resembled that various white-box criteria or functional criteria (instead of a single criterion) were adopted to generate the required equivalence classes at random.

(9) For the sake of convenience, both the 'purely-random' testing strategy and the 'random-partition' testing strategy are called 'random' testing strategy throughout the rest of this paper if no confusion occurs otherwise. Accordingly, the testing strategy defined by Algorithm A is referred to as 'adaptive' testing strategy although it can more accurately be called 'adaptive-partition' testing strategy.

4. Experiment I

This experiment is mainly concerned with the third research question identified in Section 1. It evaluates the effectiveness of the adaptive testing strategy proposed in Section 2 in comparison with the random testing strategy from a statistical perspective by using the Space program as the subject program. In this experiment, all the 38 distinct defects were injected into the Space program. Since the proposed adaptive testing strategy is mainly concerned with how to select test cases from the test suite rather than how to partition the test suite or the input domain of the software under test, validating the proposed adaptive testing strategy should not stick to any particularly given partitioning scheme. Random partitioning becomes a reasonable choice. Consequently, the test suite of 13,466 test cases[13] was partitioned into four disjoint classes at random as follows:

Class 1 comprises 3580 test cases.
Class 2 comprises 5200 test cases.
Class 3 comprises 2315 test cases.
Class 4 comprises 2371 test cases.

[13] In our case study we incidentally ignored the OLD file (non-acceptable file to the Space program) and the 30 ADL files with zero-byte (empty files). However the absence of these 31 files should not affect the validity of the results presented in this paper since the test suite we use comprises 13,466 test cases, of which 633 test cases are of zero defect detection rate.

Both the adaptive testing strategy and the random testing strategy were applied to the subject program. For the adaptive testing strategy we distinguished two scenarios[14]: $l = 100$ and $l = 200$. In this way, we had three different scenarios. The first scenario was that the random testing strategy was applied and Algorithm R was used. The second scenario was that the adaptive testing strategy was applied and Algorithm A with $l = 100$, or Algorithm A100 simply, was used. The third scenario was that the adaptive testing strategy was applied and Algorithm A with $l = 200$, or Algorithm A200 simply, was used. For the purpose of statistical analysis, we applied all the three algorithms, Algorithm R, A100 and Algorithm A200, for 10 times each.

Remarks

(1) The reason of partitioning the input domain of the subject program into four (equivalence) classes was arbitrary. The adaptive testing strategy might reduce to the random testing strategy if the input domain comprised only one class. With four classes of test cases, five parameters were estimated on-line by the genetic algorithm.
(2) The reason for choosing $l = 100$ and $l = 200$ was a bit arbitrary. If we used the whole history of testing data for the genetic algorithm to estimate and update the required parameters in the adaptive testing strategy, then there would be a severe computational complexity problem in the late phase of software testing since the whole history of testing data might comprise several thousands data points.
(3) Since various trials of software testing should be independent, we did not need to care which algorithm (Algorithm R, Algorithm A100, or 200) was performed first or next.

Ten trials were performed for each testing algorithm. In each trial of a testing algorithm we recorded the numbers of tests used between two successive observed failures. Let $b_i(k)$ be the number of tests used between the $(k-1)$th failure (exclusive) and the kth failure (inclusive) in the ith trial of a testing algorithm, and

$$bn(k) = \sum_{i=1}^{10} b_i(k)$$

Further, let

$$tn(k) = \sum_{j=1}^{k} bn(j)$$

$$\bar{b}(k) = \frac{1}{10}\sum_{i=1}^{10} b_i(k)$$

$$dn(k) = \sqrt{\frac{1}{9}\left[(b_1(k) - \bar{b}(k))^2 + \cdots + (b_{10}(k) - \bar{b}(k))^2\right]}$$

[14] Refer to footnote 10 (Section 2).

Note that $bn(k)$ denotes the total number of tests used between the $(k-1)$th failure (exclusive) and the kth failure (inclusive) in the 10 trials, $tn(k)$ the total number of tests used to detect the first kth defects in the 10 trials, $\bar{b}(k)$ the average number of tests used between the $(k-)$th failure (exclusive) and the kth failure (inclusive) in 10 trials, and $dn(k)$ the corresponding standard deviation of $\bar{b}(k)$. Obviously, the smaller the values of these measures for a given testing strategy, the better the testing strategy behaved.

Table 4.1 tabulates the total numbers of tests used in each trial. Figs. 4.1–4.3 are graphical representations of $bn(k)$, $tn(k)$ and $dn(k)$. In all the figures, the horizontal axis represents the number of observed failures. The vertical axis of Fig. 4.1 represents $bn(k)$, whereas that of Fig. 4.2

Table 4.1
Total number of test cases applied to the subject program with 38 injected defects in a single testing process using Algorithms R, A200 and A100

n	$\sum(n)$ for Algorithm R	$\sum(n)$ for Algorithm A200	$\sum(n)$ for Algorithm A100
1	1021	1456	897
2	766	343	1105
3	2939	2038	872
4	2218	1279	1060
5	1409	755	632
6	3737	2903	2639
7	2431	2998	282
8	1585	2413	1230
9	1786	2145	2279
10	4535	1602	2183
$\bar{\Sigma}$	2242.70	1793.20	1317.90
D_Σ	1202.68	872.20	778.64

$$\sum(n) = \sum_{k=1}^{36} b_n(k)$$

$$\bar{\Sigma} = \frac{1}{10}\left(\sum(1) + \cdots + \sum(10)\right)$$

$$D_\Sigma = \sqrt{\frac{1}{9}\left[\left(\sum(1) - \bar{\Sigma}\right)^2 + \cdots + \left(\sum(10) - \bar{\Sigma}\right)^2\right]}$$

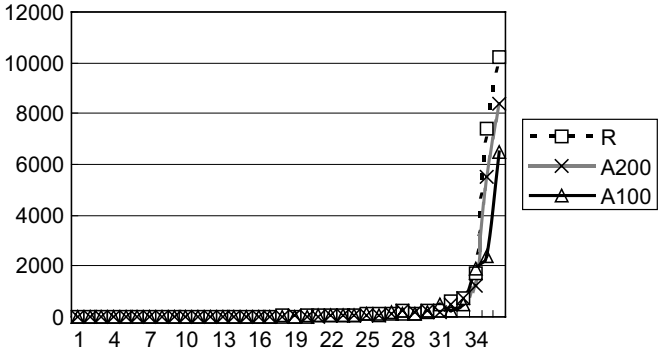

Fig. 4.1. Adaptive testing versus random testing. Horizontal axis: number of observed failures. Vertical axis: $bn(k)$ or the total number of tests used between the $(k-1)$th failure (exclusive) and the kth failure (inclusive) in the 10 trials.

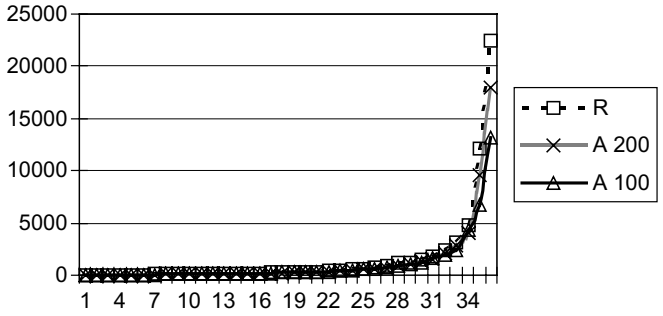

Fig. 4.2. Adaptive testing versus random testing. Horizontal axis: number of observed failures. Vertical axis: $tn(k)$ or the total number of tests used to detect the first kth defects in the 10 trials.

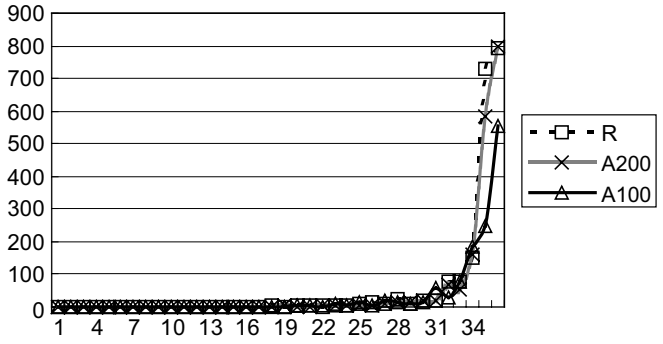

Fig. 4.3. Adaptive testing versus random testing. Horizontal axis: number horizontal axis: number of observed failures. Vertical axis: $dn(k)$ or the corresponding standard deviation of $\bar{b}(k)$.

represents $tn(k)$. $dn(k)$ is represented by the vertical axis of Fig. 4.3.

5. Experiment II

This experiment is mainly concerned with the fourth research question identified in Section 1 and tries to examine the effects of partitioning on the behavior of the adaptive testing strategy. Up to this point there was no theoretical justification how many classes the test suite should be partitioned into. Intuitively, the number of classes should have impacts on the effectiveness of the adaptive testing strategy. An extreme is that there is only one class of test cases and in this way the adaptive testing strategy reduces to the purely-random testing strategy and thus does not take any advantage of on-line adaptation or learning. However if there are too many (200, say) classes of test cases, then parameter estimation in the adaptive testing strategy is computationally intensive and may lead the adaptive testing strategy to selecting a non-optimal class since it is not easy to estimate many parameters accurately and distinguish among many classes. So, there may be an "optimal" number of equivalence classes of test cases for the adaptive testing strategy. On the other hand, the size of a class may also affect the effectiveness of a testing strategy. The purpose of this experiment was to examine how the number and the sizes of classes of test cases might affect the effectiveness of a testing strategy.

As in Experiment I, in this experiment we employed the test suite of 13,466 test cases. These test cases were then partitioned into several classes of various sizes to examine if the adaptive testing strategy could still outperform the random testing strategy under various scenarios. The Space program contained 38 distinct defects at the beginning of software testing and the adaptive testing strategy followed Algorithm A200 (Algorithm A with $l = 200$).

5.1. Case I

In this case we performed experiment to examine if the sizes of classes of test cases could affect the effectiveness of a testing strategy. We distinguished two scenarios. In the first scenarios the 13,466 test cases were partitioned into four classes at random with nearly the same size:

Class 1 comprises 3367 test cases.
Class 2 comprises 3366 test cases.
Class 3 comprises 3367 test cases.
Class 4 comprises 3366 test cases.

We denote the resulting input domain as ID1, where test cases were evenly distributed over the classes. In the second scenario the 13,466 test cases were partitioned into four classes at random such that the classes were different in size. We say the test cases are non-evenly distributed over the classes.

Class1 comprises 3580 test cases.
Class2 comprises 5200 test cases.
Class3 comprises 2315 test cases.
Class4 comprises 2371 test cases.

The resulting input domain is simply denoted as ID2.[15] Table 5.1 and Figs. 5.1–5.3 show the testing results. As in experiment I, each testing algorithm was repeated for 10 times or trials for each input domain.

Here we note that for the purely-random testing strategy the test suite is not partitioned or it contains only one equivalence class and test cases are selected in accordance with a uniform or non-uniform probability distribution or test profile. Alternatively, we can say that for the purely-random testing algorithm the test suite is partitioned into 13,466 equivalence classes, each containing a single distinct test case. Then we can reasonably say that for ID1 testing Algorithm R is equivalent to the purely-random testing strategy with a uniform test profile since the test cases are evenly distributed over the equivalence

[15] This input domain was exactly the one adopted in Experiment I. We reproduce the experimental results of Section 4 for the sake of the readability of the paper.

Table 5.1
Total number of test cases applied to the subject program with 38 injected defects in a single testing process using Algorithm R and Algorithm A200 under input domains ID1 and ID2

n	$\sum(n)$ for Algorithm R, ID1	$\sum(n)$ for Algorithm A200, ID1	$\sum(n)$ for Algorithm R, ID2	$\sum(n)$ for Algorithm A200, ID2
1	4095	687	1021	1456
2	1318	3762	766	343
3	698	4663	2939	2038
4	1687	2988	2218	1279
5	3600	765	1409	755
6	794	3896	3737	2903
7	1688	1788	2431	2998
8	3960	917	1585	2413
9	3960	2174	1786	2145
10	3960	2962	4535	1602
$\bar{\Sigma}$	2576.00	2460.20	2242.70	1793.20
D_Σ	1451.67	1417.04	1202.68	872.20

$$\sum(n) = \sum_{k=1}^{36} b_n(k)$$

$$\bar{\Sigma} = \frac{1}{10}\left(\sum(1) + \cdots + \sum(10)\right)$$

$$D_\Sigma = \sqrt{\frac{1}{9}\left[\left(\sum(1) - \bar{\Sigma}\right)^2 + \cdots + \left(\sum(10) - \bar{\Sigma}\right)^2\right]}$$

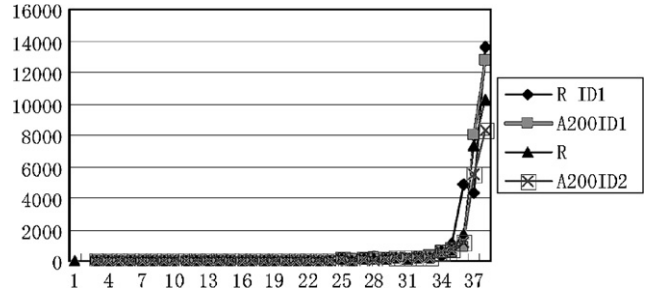

Fig. 5.1. Adaptive testing versus random testing under input domains IDI and ID2. Horizontal axis: number of observed failures. Vertical axis: $bn(k)$ or the total number of tests used between the $(k-1)$th failure (exclusive) and the kth failure (inclusive) in the 10 trials.

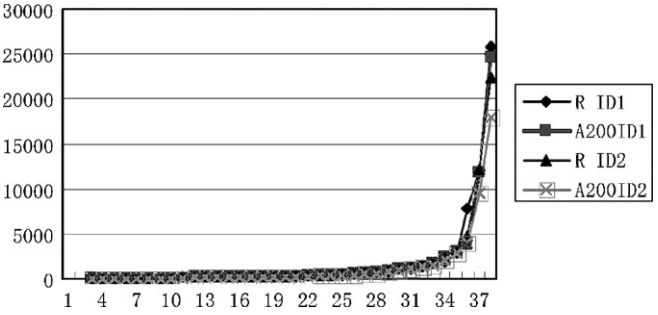

Fig. 5.2. Adaptive testing versus random testing under input domains ID1 and ID2. Horizontal axis: number of observed failures. Vertical axis: $tn(k)$ or the total number of tests used to detect the first kth defects in the 10 trials.

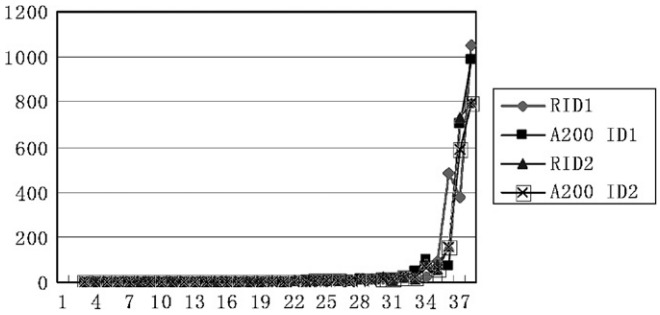

Fig. 5.3. Adaptive testing versus random testing under input domains ID1 and ID2. Horizontal axis: number of observed failures. Vertical axis: $dn(k)$ or the corresponding standard deviation of $\bar{b}(k)$.

classes. For ID2 testing Algorithm R is equivalent to the purely-random testing strategy with a non-uniform test profile.

5.2. Case II

In order to examine the impacts of the number of classes of test cases on the effectiveness of the adaptive testing strategy, we created two additional input domains. In input domain ID3 the 13,466 test cases were partitioned into five classes:

Class 1 comprises 2693 test cases.
Class 2 comprises 2693 test cases.
Class 3 comprises 2693 test cases.
Class 4 comprises 2693 test cases.
Class 5 comprises 2694 test cases.

In input domain ID4 the 13,466 test cases were partitioned into seven classes:

Class 1 comprises 1924 test cases.
Class 2 comprises 1924 test cases.
Class 3 comprises 1923 test cases.
Class 4 comprises 1924 test cases.
Class 5 comprises 1924 test cases.
Class 6 comprises 1923 test cases.
Class 7 comprises 1924 test cases.

In both ID3 and ID4 test cases were evenly distributed over the classes. As in Case I of Section 5.1, we applied Algorithm R and A200 to the Space program with 38 distinct defects. For the sake of comparison, we reproduced the testing results of testing Algorithm A200 with ID1. Table 5.2 and Figs. 5.4–5.6 show the testing results.

6. Analyses of experimental results

6.1. Analysis of Experiment I

- Fig. 4.1 suggests that no matter which testing algorithm is applied, defects become more and more difficult to be

Table 5.2
Total number of test cases applied to the subject program with 38 injected defects in a single testing process using Algorithm A200 under input domains ID1, ID3 and ID4

n	$\sum(n)$ for Algorithm A200, ID1	$\sum(n)$ for Algorithm A200, ID3	$\sum(n)$ for Algorithm A200, ID4
1	687	686	2276
2	3762	515	636
3	4663	2298	2525
4	2988	1101	1044
5	765	527	1921
6	3896	4449	2338
7	1788	941	1152
8	917	4886	1854
9	2174	482	380
10	2962	851	3002
$\bar{\Sigma}$	2460.20	1673.60	1712.80
D_Σ	1417.04	1666.86	868.63

$$\sum(n) = \sum_{k=1}^{36} b_n(k)$$

$$\bar{\Sigma} = \frac{1}{10}\left(\sum(1) + \cdots + \sum(10)\right)$$

$$D_\Sigma = \sqrt{\frac{1}{9}\left[\left(\sum(1) - \bar{\Sigma}\right)^2 + \cdots + \left(\sum(10) - \bar{\Sigma}\right)^2\right]}$$

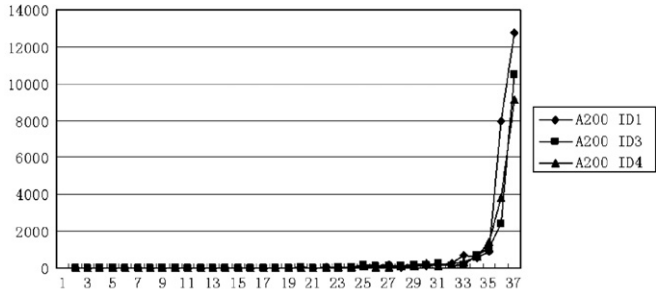

Fig. 5.4. Adaptive testing versus random testing under input domains ID1, ID3 and ID4. Horizontal axis: number of observed failures. Vertical axis: $bn(k)$ or the total number of tests used between the $(k-1)$ th failure (exclusive) and the kth failure (inclusive) in the 10 trials.

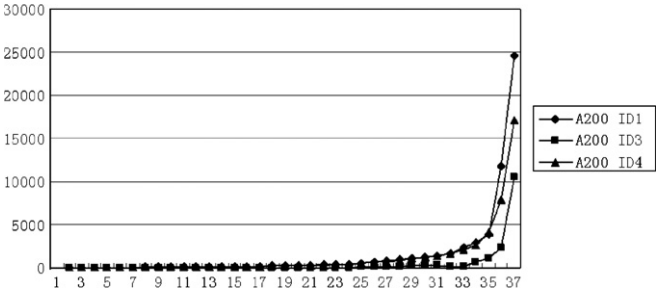

Fig. 5.5. Adaptive testing versus random testing under input domains ID1, ID3 and ID4. Horizontal axis: number of observed failures. Vertical axis: $tn(k)$ or the total number of tests used to detect the first kth defects in the 10 trials.

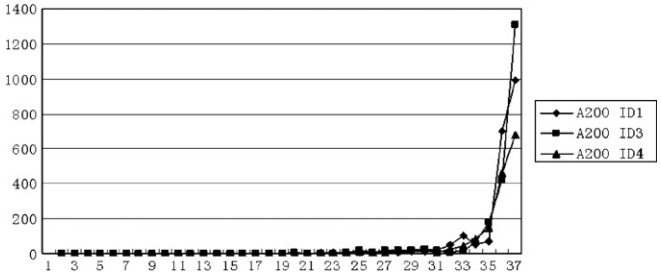

Fig. 5.6. Adaptive testing versus random testing under input domains ID1, ID3 and ID4. Horizontal axis: number of observed failures. Vertical axis: $dn(k)$ or the corresponding standard deviation of $\bar{b}(k)$.

detected as testing proceeds, or defects are not equally detectable. Actually, we can calculate the mean number of tests used to detect a new defect. Let

$$f(k) = \frac{bn(k)}{(N-k+1)\times 10} = \frac{bn(k)}{(38-k+1)\times 10} \qquad (6.1)$$

$f(k)$ represents the contribution per defect remaining in the software under test to the number of tests used to reveal the k th failure from the $(k-)$th failure in a single process of software testing. Fig. 6.1 shows the behavior of $f(k)$ graphically.[16] We can see that $f(k)$ dramatically grows in the late phase of software testing. So, Assumption (7) of Section 2 does not hold in practice and it is reasonable to treat the related software parameters as time-varying variables in on-line parameter estimations to compensate the drawbacks of the assumption. This also justifies the rationale of fixed-memory feedback.

- Fig. 4.3 shows that no matter which testing algorithm is used, the standard deviation of the number of tests used to reveal a new failure grows rapidly, and this is particularly true for the random testing strategy. This means that as the testing proceeds, the behavior of a testing strategy deteriorates and demonstrates large fluctuations.
- In comparison with the random testing strategy, the adaptive testing strategy uses fewer tests to detect and remove a given number of injected defects. This can be observed from Table 4.1. For example, in each of the 10 trials of software testing, Algorithm R uses 2242.70 tests on average. This number is 1793.20 and 1317.90 for Algorithm A200 and 100, respectively. Note $\left|\frac{2242.70-1793.20}{2242.70}\right| \times 100\% \approx 20.04\%$, $\left|\frac{2242.70-1317.90}{2242.70}\right| \times 100\% \approx 41.24\%$. The adaptive testing strategy noticeably outperforms the random testing strategy.

[16] Here we strictly follow Eq. (6.1) to calculate the values of $f(k)$. In practice the mean number of test cases used to detect a new defect is at least 1 unless one test case can detect more than one defect.

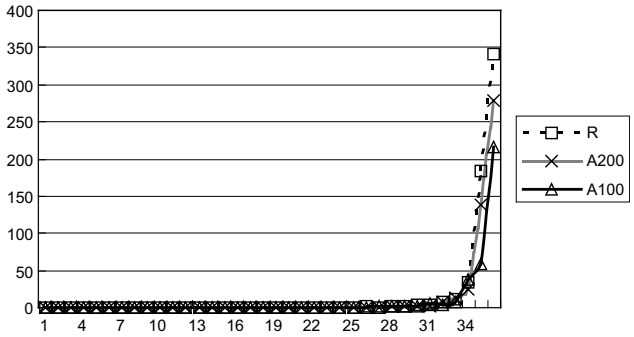

Fig. 6.1. Testing Algorithms R, A200 and A100 under Experiment I. Horizontal axis: number of observed failures. Vertical axis: $f(k)$ or the contribution per defect remaining the number of tests used to reveal the kth failure from the $(k-1)$th failure.

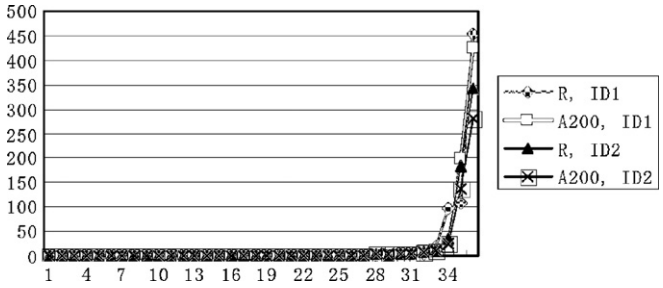

Fig. 6.2. Testing Algorithms R and A200 under Experiment II, Case I. Horizontal axis: number of observed failures. Vertical axis: $f(k)$ or the contribution per defect remaining the number of tests used to reveal the kth failure from the $(k-1)$th failure.

- From Table 4.1 we can see that in the 10 trials of software testing, testing Algorithm R demonstrates a standard deviation of 1202.68, whereas this number is 872.20 and 778.64 for testing Algorithm A200 and 100, respectively. Note standard deviation represents stability or dispersion of a testing algorithm in some sense, and $\left|\frac{1202.68-872.20}{1202.68}\right| \times 100\% \approx 27.48\%$, $\left|\frac{1202.68-778.64}{1202.68}\right| \times 100\% \approx 35.26\%$, we can roughly say that the adaptive testing strategy is about 30% more stable than the random testing strategy.
- Testing Algorithm A100 behaves slightly better than testing Algorithm A200. In each of the 10 single processes of software testing, testing Algorithm A100 uses 1317.90 test cases on average, whereas this number is 1793.20 for testing Algorithm A200. Also, testing Algorithm A100 is more stable. The standard deviation of the number of tests used in a single process for testing Algorithm A100 to detect and remove the 36 injected defects is 778.64, whereas this number is 872.20 for testing Algorithm A200. This means that the size of memory used for feedback in the adaptive testing strategy does affect the effectiveness of the adaptive testing strategy. However how the size of feedback memory affects the behavior of the adaptive testing strategy is a topic of further investigation.

6.2. Analysis of Experiment II: Case I

- Fig. 6.2 shows the behavior of $f(k) = \frac{bn(k)}{(N-k+1)\times 10} = \frac{bn(k)}{(38-k+1)\times 10}$ under case I of experiment II. It demonstrates a similar scenario that is observed in Experiment I.
- Distribution of test cases over equivalence classes makes impacts on the behavior of the random testing strategy. Figs. 5.1–5.3 show clearly that both testing Algorithms R and $A200$ behave differently under input domain ID1 and ID2. More specifically, from Table 5.1 we have the following for the behavior of $\bar{\Sigma}$ and D_{Σ} for the random testing strategy

$$\left|\frac{2576.00-2242.70}{2576.00}\right| \times 100\% \approx 12.94\%$$
$$\left|\frac{1451.67-1202.68}{1451.67}\right| \times 100\% \approx 17.15\%$$

That is, ID2 is better than ID1 in terms of the average number of tests used in a trial as well as in terms of the corresponding standard deviation. For the adaptive testing strategy, we have

$$\left|\frac{2460.20-1793.20}{2460.20}\right| \times 100\% \approx 27.11\%$$
$$\left|\frac{1417.04-872.20}{1417.04}\right| \times 100\% \approx 38.45\%$$

ID2 outperforms ID1 noticeably in terms of both the average number of tests used in a trial and the corresponding standard deviation. Overall, it looks that non-even distribution of test cases helps a testing algorithm to improve the performance and its stability.

- We may explain the effects of non-even distribution of test cases as follows. If test cases are evenly distributed over equivalence classes, we can imagine that each equivalence class has a roughly equal capability of defect detection. Non-even distribution of test cases over the equivalence classes of ID2 makes these equivalence classes have different capabilities of defect detection. The capability of defect detection of some of equivalence class is enhanced as a result of the transition from even distribution to non-even distribution of test cases. For the adaptive testing strategy, the on-line scheme of adaptation or learning makes the strategy tend to select test cases from equivalence classes of top rates of defect detection. Therefore ID2 is preferred to ID1 for the adaptive testing strategy. For the random testing strategy, ID2 may have advantage over ID1 since ID2 assumes a non-uniform test profile for the purely-random testing strategy. However whether a non-uniform test profile performs better than a uniform test profile or vice versa may largely depend on application.

- For both input domains ID1 and ID2, the adaptive testing strategy outperforms the random testing strategy. This can be justified by the values of $\bar{\Sigma}$ and D_{Σ} tabulated in Table 5.1. For ID1, we have

$$\left|\frac{2576.00 - 2460.20}{2576.00}\right| \times 100\% \approx 4.50\%$$
$$\left|\frac{1451.67 - 1417.04}{1451.67}\right| \times 100\% \approx 2.39\%$$

  That is, the adaptive testing strategy is slightly better than the random testing strategy. For ID2, these two values become as the following:

$$\left|\frac{2242.70 - 1793.20}{2242.70}\right| \times 100\% \approx 20.04\%$$
$$\left|\frac{1202.68 - 872.20}{1202.68}\right| \times 100\% \approx 27.48\%$$

  The input domains have significant effects on the behavior of software testing strategies. ID2 is preferred to ID1 again.
- In summary, the adaptive testing strategy with input domain ID2 (non-even distribution of test cases) behaves better than the random testing strategy with input domain ID2, which in turn behaves better than the random testing strategy with input domain ID1 (even distribution of test cases). Recall that the random testing strategy with input domain ID1 is equivalent to conventional random testing strategy (refer to Section 6.1), we can reasonably say that *the adaptive testing strategy can noticeably outperform the conventional random testing strategy*.

6.3. Analysis of Experiment II: Case II

- Fig. 6.3 shows the behavior of $f(k)$ under case II of experiment II. It demonstrates a similar scenario that is observed in Experiment I and Case I of Experiment II.
- Table 5.2 and Figs. 5.4–5.6 show a mixed pattern about ID1, ID3 and ID4. Specifically, from Table 5.2 we know that ID3 outperforms ID1 in terms of the average total number of tests used in a single trial,

$$\left|\frac{2460.20 - 1673.60}{2460.20}\right| \times 100\% \approx 31.97\%$$

  However ID1 outperforms ID3 in terms of the corresponding standard deviation,

$$\left|\frac{1666.86 - 1417.04}{1666.86}\right| \times 100\% \approx 14.99\%$$

  Further, ID4 outperforms ID3 in terms of the average total number of tests used in a single trial,

$$\left|\frac{1712.80 - 1673.60}{1712.80}\right| \times 100\% \approx 2.29\%$$

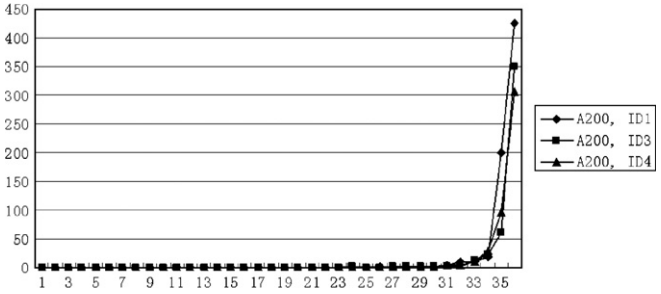

Fig. 6.3. Testing Algorithms R and A200 under Experiment II, Case II. Horizontal axis: number of observed failures. Vertical axis: $f(k)$ or the contribution per defect remaining the number of tests used to reveal the kth failure from the $(k-1)$th failure.

  ID3 outperforms ID4 in terms of the corresponding standard deviation,

$$\left|\frac{1666.86 - 868.63}{1666.86}\right| \times 100\% \approx 47.89\%$$

  That is, the number of equivalence classes has mixed impacts on the behavior of a testing algorithm. Having more equivalence classes can hardly have positive impact on both the average number of tests used in a single trial and the corresponding standard deviation. However if we can ignore the differences less than 10% in terms of relative errors, then we can roughly say that having more equivalence classes helps to improve the behavior of a testing algorithm. Of course, a comprehensive case study should exclusively be carried out to examine how the number of equivalence classes affects the behavior of the adaptive testing strategy.
- Compare the testing results tabulated in Table 5.2 with those tabulated in Table 5.1. We can see that testing Algorithm A200 with ID3 and ID4 outperforms testing Algorithm R with ID1 as well as ID2 in terms of $\bar{\Sigma}$. Specifically, testing Algorithm A200 with ID4 significantly outperforms testing Algorithm R with ID1 in terms of both the average total number of test cases used in a single trial and the corresponding standard deviation

$$\left|\frac{2576.00 - 1712.80}{2576.00}\right| \times 100\% \approx 33.51\%$$
$$\left|\frac{1451.67 - 868.63}{1451.67}\right| \times 100\% \approx 40.16\%$$

  Note that testing Algorithm R with ID1 corresponds to purely-random testing strategy.

6.4. General observations

Overall, we can have the following observations:

- No matter what testing strategy is applied or what testing scenario is considered, in general software testing

becomes more and more difficult as more and more defects are detected and removed in the sense that the number of tests required for revealing a new failure is increasing. This is particularly true in the late stage of software testing.
- Behavior of a testing strategy becomes less and less stable as software testing proceeds in the sense that the standard deviation of the number of tests required for revealing a new failure is increasing. This is particularly true in the late stage of software testing.
- Software defects tend to be non-equally detectable. It looks more reasonable to treat the probability of a defect being detected as a time-varying variable than otherwise (as a constant).
- The adaptive testing strategy with fixed-memory feedback can noticeably outperform the random testing strategy in terms of the average number of tests used for detecting and removing a certain number of defects as well as the corresponding standard deviation. The adaptive testing strategy may use up to 40% fewer tests and in the meantime, is up to 40% more stable than the random testing strategy. Various factors make contribution to the behavior of a testing strategy.
- The size of feedback memory used for on-line parameter estimation in the adaptive testing strategy does affect the behavior of the adaptive testing strategy. Larger size of feedback memory may or may not lead to better behavior of the adaptive testing strategy. This implies that software testing data may or may not age. However how the software testing data age is a topic of further investigation.
- How test cases are distributed over the equivalence classes of the input domain affects the behavior of the adaptive testing strategy as well as that of the random testing strategy. Non-even distribution should be encouraged for the adaptive testing strategy as well as for the random testing strategy.
- The number of equivalence classes in an input domain may have mixed impacts on the behavior of the adaptive testing strategy in terms of the average number of tests used for detecting and removing a certain number of defect as well as the corresponding standard deviation. However the adaptive testing strategy consistently outperforms the random testing strategy even if different number of equivalence classes is adopted.
- It is true that some equivalence classes are more capable of detecting defects than others. This is an intrinsic characteristic and objective basis of software for the adaptive testing strategy to demonstrate its muscle. The adaptive testing strategy tends to select equivalence classes of higher defect detection rates.
- It is true that some test cases are similar and some test cases are dissimilar in terms of defect detection rate. Similar test cases should be clustered and put into equivalence classes. Testing results of a test case can be used to infer information and assess the defect detection ability for its similar test cases.
- The major factor that makes the adaptive testing strategy outperform the random testing strategy is that the former follows a closed-loop (feedback) philosophy, whereas the latter follows an open-loop philosophy. The rationale for following the closed-loop philosophy can be justified from various perspectives such as
  (1) A test case that detects a defect may be able to detect other defects that have coupling effects with the defect detected already.
  (2) Similar test cases may detect similar defects.
  (3) Equivalence classes have non-equal defect detection rates.
  (4) Defects tends to cluster.Overall, the closed-loop philosophy may behave better than the open-loop philosophy even if the simplifying models (or unrealistic models in some sense) of the software under test are adopted.

In summary, the four research questions identified in Section 1 can be answered as follows:

(1) In order to circumvent the unrealistic assumption in the software testing process, to avoid aging testing data and to reduce the on-line computational burden, partial history of testing data should be adopted for feedback to replace the whole history of testing data. The advantages of adopting partial history over adopting whole history of testing data can be justified by the widely accepted intuitive observation that software defects tend to be non-equally detectable and the probability of a defect being detected is time-varying during testing. This observation is validated by the experimental results presented in this paper.[17]
(2) The adaptive testing strategy is effective for real software programs. The unrealistic assumption that the initial number of defects remaining in the software under test is known should not be a hurdle for this conclusion. Note that in adaptive or random testing, the unrealistic assumption was only used for determining when software testing should stop. Suppose that the initial number of defects is unknown, then other stopping criteria or testing goals can be adopted. An example testing goal is to detect as many distinct defects as possible in 1000 tests. Then software testing must be stopped after 1000 tests. The unrealistic assumption about the initial number of software defects by no means suggests that the

[17] For the adaptive testing strategy which adopts the whole history of testing data for feedback, Algorithm A presented in Section 3 still works by assigning a huge value to the parameter l. We intentionally avoid presenting related experimental results. In the subsequent two papers of ours, we will examine how the parameter l affects the behavior of adaptive testing and how to determine an "optimal" value for l. See footnote 10 in Section 2. In this paper we only show that the adaptive testing strategy with an appropriate value of l can significantly outperform the random testing strategy.

proposed adaptive testing strategy cannot be applied in practice. The proposed testing strategy can be applied as extensively as the random testing strategy.
(3) The given test suite or the input domain of the software under test should be partitioned in such a way that test cases are non-evenly distributed over equivalence classes for adaptive testing. However it remains unclear how many equivalence classes the input domain of the software under test should be partitioned.

6.5. Future works

Several specific problems have not been addressed in this paper. They should be investigated in the future. However they are not trivial and each of them requires one or more separate research papers.

(1) A detailed analysis of the computational complexity imposed by the adaptive testing strategy should be given to reveal the trade-offs involved. It may not be useful to spend a lot of time and space to compute an optimal test selection if we could execute several sub-optimal tests while computing the optimal one. Fortunately, Footnote 8 suggests that the problem of time complexity should not be a hurdle for the adaptive testing strategy to be applied in practice.
(2) Some software systems under test provide only pass/fail information while others could provide also some other type of observations useable in test selection. It is not clear at this stage how the related information can be employed to improve the adaptive testing strategy. Note that the adaptive testing strategy presented in this paper uses only pass/fail information to improve the testing process.
(3) Software defects may or may not be correlated with each other (Jin et al., 2005). The existence of one particular defect may mask another defect, making the latter non-detectable. On the other hand, the existence of one particular defect may be the prerequisite for another defect to be detectable. An interesting question is how defect correlation affects the effectiveness of the adaptive testing strategy.
(4) The length of the history data or the amount of feedback information is a important parameter for the adaptive testing strategy. Intuitively, the amount of feedback information should not too much. An extreme is that all the history data are employed. On the other hand, the amount of feedback information should not too small either. Another extreme is no feedback information is employed. There should be an optimal amount of feedback information for the adaptive testing strategy.
(5) Besides the random testing strategies discussed in the preceding sections, other existing testing strategies should also be examined in comparison with the adaptive testing strategy. This will lead to better understanding how the adaptive testing strategy make advance in the state-of-the-art of software testing.

6.6. Threats to the validity of the observations

- The major threat to the validity of the observation that the adaptive testing strategy may noticeably outperform the random testing strategy is that we have not obtained the upper bound of the performance of the random testing strategy including the purely-random testing strategy and the random-partition testing strategy. We note that there have been a lot of researches and controversies about random testing and partition testing (Weyuker and Jeng, 1991; Duran and Ntafos, 1984; Hamlet and Taylor, 1990; Chen and Yu, 1996; Gutjahr, 1999; Ntafos, 1998). Since both the adaptive testing strategy and the random testing strategy are statistical in nature, we may imagine that under some special scenarios the random testing strategy may outperform the adaptive testing strategy.
- Our observations are mainly drawn from the case study presented in this paper. However there are weaknesses in the case study. Only 10 trials are carried for each scenario. More trials may help to improve the statistical accuracy of the observations we draw.
- The subject program used in our case study is a C language program whose output is uniquely determined by the current input and is independent of the history of input. However many software systems demonstrate different scenarios. The outputs of these software systems depend not only on the current input, but also the history of input. New large scale software systems should be used to test the observations drawn in this paper.
- For large scale software systems there are often tens of equivalence classes rather than 4 or 7 equivalence classes as adopted in our case study. This may be a factor that affects the accuracy of the observations drawn in this paper.
- Assumption (7) presented in Section 2 can always be under attack since it deviates from the reality. A more reasonable assumption may be required in the future.

7. Concluding remarks

Up to this point we have shown how to improve the adaptive testing strategy proposed in our previous work by using fixed-memory feedback for on-line parameter estimations. The underlying motivation is to circumvent the drawbacks of the assumption that all remaining defects are equally detectable at constant rate and to reduce the induced computational complexity of on-line parameter estimations. The case study with the Space program demonstrates that the improved adaptive testing strategy can really work in practice and may noticeably outperform

the purely-random testing strategy and random-partition testing strategy (or collectively, the random testing strategy) in terms of the number of tests used to detect and remove a given number of defects in a single process of software testing and the corresponding standard deviation. This has vast potentials in theory as well as in practice. The adaptive testing strategy is theoretically sound in the sense that it is based on existing control theory. It further justifies and strengthens the feasibility of the idea of software cybernetics which is supposed to explore the interplay between software and control (Cai et al., 2003). It is possible for us to make software testing processes in particular, and software engineering processes in general, more rigorous and theoretically justified at minor cost. In practice this paper shows that the improved adaptive testing strategy can be applied conveniently and even automated as long as the test suite is given and partitioned a priori, and the test oracle is not at odd. It may contribute to testing large-scale software systems more than to testing small-scale software systems.

On the other hand, there are many topics deserving further investigation. For the random testing strategy, a comprehensive study should be carried out to examine how partitioning of input domain affects the performance of it. Theoretical analyses are also required to justify the empirical observations presented in this paper. For the new adaptive testing strategy proposed in this paper, further studies should examine how the amount of fixed-memory feedback and the number of equivalence classes affect the behavior of software testing. For software cybernetics, other strategies of adaptive testing should be developed to address different testing goals such as maximizing the number of detected defects in a given number of tests.

References

Cai, K.Y., 1998. Software Defect and Operational Profile Modeling. Kluwer Academic Publishers.

Cai, K.Y., 2000. Towards a conceptual framework of software run reliability modeling. Information Sciences 126, 137–163.

Cai, K.Y., 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. Information and Software Technology 44, 841–855.

Cai, K.Y., Cangussu, J.W., DeCarlo, R.A., Mathur, A.P., 2003. An overview of software cybernetics. In: Proc. the 11th International Workshop on Software Technology and Engineering Practice. IEEE Computer Society Press, pp. 77–86.

Cai, K.Y., Li, Y.C., Ning, W.Y., 2005. Optimal software testing in the setting of controlled Markov chains. European Journal of Operational Research 162 (2), 552–579.

Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M., 2005. Model-Based testing of object-oriented reactive systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, May .

Cangussu, J.W., DeCarlo, R.A., Mathur, A.P., 2001. Feedback Control of the software test process through measurements of software reliability. In: Proc. 12th International Symposium on Software Reliability Engineering.

Cangussu, J.W., DeCarlo, R.A., Mathur, A.P., 2002. A formal model of the software test process. IEEE Transactions on Software Engineering 28 (8), 782–796.

Chen, T.Y., Yu, Y.T., 1996. On the expected number of failures detected by subdomain testing and random testing. IEEE Transactions on Software Engineering 22 (2), 109–119.

Chen, T.Y., Tse, T.H., Yu, Y.T., 2001. Proportional sampling strategy: a compendium and some insights. Journal of Systems and Software 58, 65–81.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. IEEE Computer 11, 34–41.

Duran, J.E., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering SE-10 (4), 438–444.

Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering 28 (2), 159–182.

Gutjahr, W.J., 1999. Partition testing vs. random testing: the influence of uncertainty. IEEE Transactions on Software Engineering 25 (5), 661–674.

Hamlet, D., Taylor, R., 1990. Partition testing does not inspire confidence. IEEE Transactions on Software Engineering 16 (12), 1402–1411.

Hernandez-Lerma, O., 1989. Adaptive Markov Control Processes. Springer-Verlag.

Jin, T., Jiang, C.H., Hu, D.B., Bai, G.C., Cai, K.Y., 2005. An approach for detecting correlated software defects. Journal of Software 16 (1), 17–28.

Lyu, M.R. (Ed.), 1996. Handbook of Software Reliability Engineering. McGraw-Hill.

Malaiya, Y.K., 1995. Antirandom testing: getting the most out of black-box testing. In: Proc. 6th International Symposium on Software Reliability Engineering, pp. 86–95.

Myers, G.J., 1979. The Art of Software Testing. John Wiley & Sons.

Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W., 2004. Optimal Strategies for Testing Nondeterministic Systems. In: Proc. the International Symposium on Software Testing and Analysis.

Ntafos, S., 1998. On Random and Partition Testing. In: Proc. International Symposium on Software Testing and Assessment, pp. 42–48.

Pacheco, C., Ernst, M.D., 2005. Eclat: automatic generation and classification of test inputs. In: Proc. the 19th European Conference on Object-Oriented Programming.

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27 (10), 929–948.

Veanes, M., Campbell, C., Schulte, W., Kohli, P., 2005. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-05, Microsoft Research, January.

Vokolos, F.I., Frankl, P.G., 1998. Empirical evaluation of the textual differencing regression testing technique. In: Proc. the International Conference on Software Maintenance, November, pp. 44–53.

Weyuker, E.J., Jeng, B., 1991. Analyzing partition testing strategies. IEEE Transactions on Software Engineering 17 (7), 703–711.

Whittaker, J.A., Poore, J.H., 1993. Markov analysis of software specifications. ACM Transactions on Software Engineering and Methodology 2 (1), 93–106.

Xie, M., 1991. Software Reliability Modeling. World Scientific.

Xie, T., Notkin, D., 2003. Tool-assisted unit-test generation and selection based on operational abstractions. In: Proc. the 18th Annual International Conference on Automated Software Engineering, pp. 40–48.

Yin, H., Lebne-Dengel, Z., Malaiya, Y.K., 1997. Automatic test generation using checkpoint encoding and antirandom testing. In: Proc. 8th International Symposium on Software Reliability Engineering, pp. 84–94.

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 City University of Hong Kong, Swinburge University of Technology (Australia), University of Technology, Sydney (Australia), and Purdue University (USA). He has published over 35 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* (Tshinghua University Press, Beijing, 1995, in Chinese). He serves on the editorial board of the international journal *Fuzzy Sets and Systems* and is the editor of the *Kluwer International Series on Asian Studies in Computer and Information Science* (http://www.wkap.nl/prod/s/ASIS). He served as program committee co-chair for the Fifth International Conference on Quality Software (Melbourne, Australia, September 2005), the First International Workshop on Software Cybernetics (Hong Kong, September 2004), and the Second International Workshop on Software Cybernetics (Edinburgh, UK, July 2005), general co-chair for the Third International Workshop on Software Cybernetics (Chicago, September 2006), and the Second International Symposium on Service-Oriented System Engineering (Shanghai, October 2006). He also served as guest editor for *Fuzzy Sets and Systems* (1996), the *International Journal of Software Engineering and Knowledge Engineering* (2006), and the *Journal of Systems and Software* (2006). His main research interests include software reliability and testing, intelligent systems and control, and software cybernetics.

Bo Gu was born in September 1982. He received his M.S. degree at the Ohio State University in 2006 and his B.E. degree at Beihang University (Beijing University of Aeronautics and Astronautics) in 2004. Now he is employed as a product development engineer in Texas Instruments Inc. at Dallas, USA.

Hai Hu was born in August 1983. He receives his B.S degree at Beihang University (Beijing University of Aeronautics and Astronautics) in 2004. He worked as a visiting scholar at the University of Texas at Dallas from 2005 to 2006. Now he is a graduate student at Beihang University under the supervision of Professor Kai-Yuan Cai. His main research topics are software testing and software cybernetics.

Yong-Chao Li was born in September 1977. He received his B.S. degree at He-Nan University of Technology in 2000 and his M.S. degree at Beihang University (Beijing University of Aeronautics and Astronautics) in 2003. Now he is employed as a product development engineer in IBM China Development Laboratory, at Shanghai.