Adaptive Software Testing in the Context of an Improved Controlled Markov Chain Model

Hai Hu, Chang-Hai Jiang and Kai-Yuan Cai[1]
Beijing University of Aeronautics and Astronautics, Beijing 100083, China
kycai@buaa.edu.cn

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 improves. Previous studies on adaptive testing rely on a simplified Controlled Markov Chain (CMC) model for software testing which employs several unrealistic assumptions. In this paper we propose a new adaptive software testing approach in the context of an improved CMC model which aims to eliminate such threats to validity. A new set of basic assumptions on the software testing process is proposed and several unrealistic assumptions are replaced by more common situations in real life software testing. The methodology of a new adaptive testing strategy is also developed and implemented. Experimental data are collected to demonstrate the effectiveness of the new methodology.

1. Introduction
The strategy used for testing a software system should be dynamic, because as testing proceeds we may gain understanding of the software under test. Adaptive testing first proposed in [6], provides a way to accomplish this by applying software cybernetics and controlled Markov chains (CMC) to software testing.

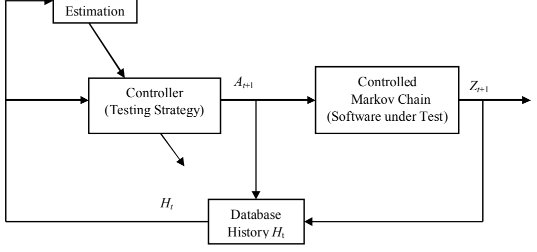
Parameters Estimation
Controller (Testing Strategy)
A_{t+1}
Controlled Markov Chain (Software under Test)
Z_{t+1}
H_t
Database History H_t

Figure 1 An Overview of Adaptive Testing

[1] The work reported here is supported by the National Science Foundation of China and Microsoft Research Asia (Grant No. 60633010) and the 863 Programme of China (Grant Nos. 2006AA01Z174 and Grant No. 2006AA01Z157).

Figure 1 gives a pictorial overview of the adaptive testing--a topic of software cybernetic which studies the interplay between software and control. The software under test is viewed as a controlled object which is modeled by a CMC, and the testing strategy serves as the corresponding controller. Together, they make up a closed-loop feedback control system.

The basic structure of the adaptive testing approach remains unchanged in this study. However, several major improvements have been carried out in the methodology to cope with the change of basic assumptions in the improved CMC model. The parameter estimation module is re-designed to estimate a parameter matrix instead of a vector. The decision making module is also revised according to the new definition for state of the software under test. And finally a new recursive algorithm is devised in order to reduce computational overhead.

The rest of this paper is organized as follows: Section 2 formulates the improved CMC model for software testing; Section 3 gives the new adaptive testing methodology; Empirical studies and data are reported in Section 4; conclusions and future research plans are included in Section 5.

2. Related Studies
This paper is related to several topics in software testing: first it is closely related to previous works on adaptive testing; Cai proposed the adaptive testing methodology in based on a fixed-memory feedback mechanism to improve the failure detecting efficiency of traditional random testing. Hu extended the adaptive testing approach to testing for software components in [6]. Other studies include how to apply adaptive testing for reliability assessment [2], or how to improve the parameter estimation process.

However the above works all based on the simplified controlled Markov chain model of software testing which requires several undesirable assumptions. [3] proposes an improved model for software reliability testing which intends to overcome these unrealistic assumptions but only simulation results are presented and it gives no further investigation on testing strategy.

The problem studied in this paper is also closely related to the test case prioritization problem [9]. Elbaum et.al reported that in regression testing feedback may play positive or negative role in test case prioritization [5]. Do et.al presented an empirical study on assessing the ability of prioritization techniques to improve the rate of fault detection of test case prioritization techniques, measured relative to mutation faults in [4]. The major difference between this work and other test case prioritization techniques is the prioritization is carried out on-the-fly as testing proceeds, which means testing history information is collected and used for future decision making.

Other related studies include defect removal and its impact on software testing. Okamura proposed a new reliability estimation method that considers defect removal [8]. This study presents a rigorous model for the defect removal process and its impact to the software under test, and developed the according methodology for testing and parameter estimation.

3. The Improved CMC Model for Software Testing

The original CMC model for adaptive testing faces several threats to validity:

- Software defects are *equally detectable*. This is *not* true in real-life testing.
- The testing process stops when a defect is detected. The defect is then removed *immediately*. However this might not be true in most testing schemes, a more commonly used defect removal is *batch debugging* which removes a number of bugs after a period of testing.

In this paper, we try to develop an improved CMC model for testing process to overcome these limitations. More specifically, the test goal is to detect and remove as many software defects as possible with a certain number of test cases.

The improved model for software testing is based on the following assumptions:

(1) The input domain or the given test suite, C, of the software under test comprises m classes of test cases, C_1, C_2, . . ., C_m, which may or may not be disjoint.

(2) The software under test contains N defects at the beginning of the testing process, in which each of the defects is not equally detectable.

(3) The software testing process terminates when M test actions have been taken, that is the maximum number of allowed actions is M.

(4) Each of the N defects is in one of three distinct states at any time: removed or absent from the software under test, undetected by any action, or detected but not removed from the software under test, symbolically, let

$$Y_t^{(k)} = \begin{cases} 0 \text{ if the } k\text{th defect has been removed} \\ \quad \text{from the software at time } t; \\ 1 \text{ if the kth defect remains undetected at time } t; \\ 2 \text{ if the kth defect is detected but not removed} \\ \quad \text{from the software at time } t. \end{cases}$$

$k = 1,2,...,N$ with $Y_0^{(1)} = Y_0^{(2)} = ... = Y_0^{(N)} = 1$

(5) The state of the software can then be denoted by $\xi_t = \left[Y_t^{(1)}, Y_t^{(2)}, ..., Y_t^{(N)}, X_t\right]$, where X_t denotes the remaining number of test actions.

(6) There are m admissible actions at each time, and there is a special action A_{m+1} which removes a certain number defects from the software.

(7) Each action detects at most one defect and incurs a cost $w_{\xi_t}(A_t)$, no matter whether it triggers a failure or not; action A_{m+1} incurs no cost. $w_{\xi_t}(A_t)$ denotes the cost of taking the i_{th} action at state ξ_t.

(8) Action A_t taken at state ξ_t gives a rebate $\sigma_{\xi_t}(A_t)$ if it triggers a defect that has never been detected, i.e., $Y_t^{(j)} = 1$. A rebate is defined as the "benefit" that a tester may receive due to the detection of a failure. Action A_{m+1} does not generate any rebate.

(9) Z_t depends only on the software state ξ_t, where the probability of a test action detecting a defect is determined by Θ:

$$\Theta = \begin{bmatrix} \theta_1^{(1)} & \theta_1^{(2)} & \cdots & \theta_1^{(N)} \\ \theta_2^{(1)} & \theta_2^{(2)} & \cdots & \theta_2^{(N)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_m^{(1)} & \theta_m^{(2)} & \cdots & \theta_m^{(N)} \end{bmatrix}$$

$\Pr\{Z_t=1 | A_t = i\} = \sum_{k=1}^{N} \theta^{(k)}{}_i$, $\Pr\{Z_t=0 | A_t = i\} = 1 - \sum_{k=1}^{N} \theta^{(k)}{}_i$;

Note that in the original CMC model, the probability of action detecting a defect is denoted by a vector $\Theta' = [\theta_1, \theta_2, \ldots, \theta_m]$.

(10) Upon a total of d new failure being revealed, the corresponding d failure-causing defects are removed immediately and instantaneously from the software under test, and no new defects are introduced.

(11) The target state of the software is $\xi_{fin} = [0,0,...,0]$, it's the absorbing state.

We have the following remarks on the above assumptions:

(1) A major distinction of the present model from the simplifying model is the extended intermediate variable Y_t which identifies the state

of a defect. By adding a new status "2" for each defect, detecting a defect does not necessarily lead to a defect removal action; the "detected but not removed" state is introduced to facilitate batch debugging which removes a number of defects at one time.

(2) Another major improvement is the extended failure detection rate matrix Θ, which indicates that defects are not equally detectable. However θ_{ij} might not be independent of each other because in real-life software testing defects are commonly found correlated.

(3) The generalized model reduces to the original CMC model [1] if $d = 1$ and $\theta_{jk} \equiv \theta_j$; $k = 1, 2, ..., N$ and $j = 1, 2, ..., m$.

(4) Assumption (9) gives the stopping criterion of the testing process. Theoretically, testing stops when all defects are detected and removed, i.e., $\xi_{fin} = [0, 0, ..., 0]$. However this is usually not possible in real life testing since N is unknown to the tester and there are always remaining bugs in a released version of software..

The above assumptions define a controlled Markov chain as shown in Figure 2. The state-transition chart is much more complicated than that of the original model because of the introduction of different failure detection rates with respect to defect. The state space increases exponentially with the number of detected defects.

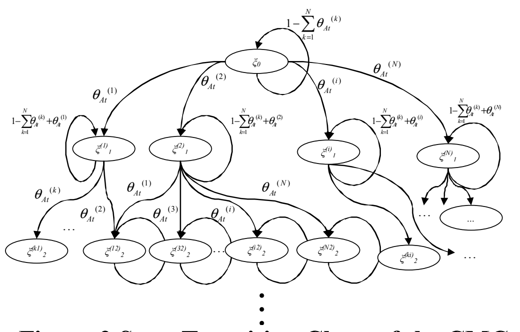

Figure 2 State Transition Chart of the CMC

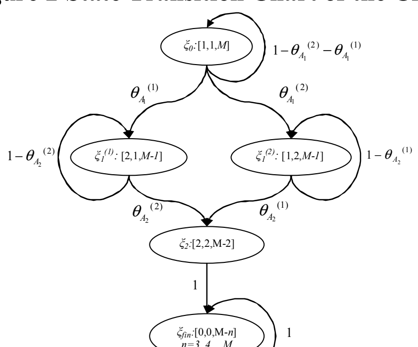

Figure 3 State Transition Chart (N=2 and d=2)

Figure 3 depicts a simple example of the Markov Chain state transition chart, when N=2 and d = 2. The problem we try to solve here is how to detect as many defects as possible within M test actions. Further let ω denote the testing strategy that is adopted in the process of software testing. ω specifies how test cases should be 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. In order to solve the problem in context with the Control Markov Chain theory, we define the expected total cost of the software testing process as:

$$J_\omega(M) = E_\omega \sum_{t=0}^{M} [W_{\xi_t}(A_t) - \theta^*_{A_t} \sigma_{\xi_t}(A_t)] \qquad (1)$$

where $J_\omega(M)$ is the total cost for all M actions and ω is the testing strategy. Our objective is to find a testing strategy that minimizes $J_\omega(M)$. Such a strategy uses M tests to maximize the overall rebate while minimizing the total cost incurred by executing these test cases.

Moreover let

$$\begin{aligned} &w^*_\xi(i) = W_\xi(i) - \theta^*_i \sigma_\xi(i) \\ &W_{\xi_j}(i) \equiv 1, \sigma_{\xi_j}(i) \equiv 1; \forall j \neq 0, \forall i \end{aligned} \qquad (2)$$

Then the corresponding testing strategy detects and removes as many defects as possible with the M testing actions, note that detecting a defect that has already been detected but not removed does not generate any rebate thus it's not encouraged by the testing strategy. Also in equation (1), θ^*_{At} is the probability of detecting a *new* defect in state ξ_t. According to assumption (1) through (9) we have

$$\theta^*(A_t = i) = \begin{cases} \sum_{k=1}^{N} \theta_i^{(k)} sign(Y_t^{(k)} - 2) sign(-Y_t^{(k)}); & i = 1, 2, ...m \\ 0; & i = m+1 \end{cases} \qquad (3)$$

$$\text{where } sign(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

From the theory of Controlled Markov Chains [6] we can conclude that there exists a deterministic stationary that minimizes $J_\omega(M)$, According to the method of successive approximation, let

$$v_{n+1}(\xi) = \min_{1 \le i \le m} \left\{ w^*_\xi(i) + \sum_{\eta \neq \xi_{fin}} q_{\xi\eta}(i) v_n(\eta) \right\} \qquad (4)$$

$$v(\xi) = \lim_{n \to \infty} v_n(\xi) \qquad (5)$$

We have the following proposition:

$$v(\xi_t)=\begin{cases}0; & \text{if } X_t=0\\ \min\limits_{1\le i\le m}\left\{w^*_{\xi_t}(i)\right\}; & \text{if } X_t=1\\ \min\limits_{1\le i\le m}\left\{w^*_{\xi_t}(i)+\theta^*_i v\left({\xi_{t+1}}^{(k)}\right)+\left(1-\theta^*_i\right)v\left({\xi_{t+1}}^{(0)}\right)\right\}; & \text{if } X_t>1\end{cases} \tag{6}$$

whereas ${\xi_{t+1}}^{(k)}$ denotes the subsequent state of state ξ_t if test action A_t detects the *k*th defect, and ${\xi_{t+1}}^{(0)}$ denotes the subsequent state of ξ_t if A_t detects no defect at all.

The above proposition clearly indicates how to test the software with the limited *M* test actions: at each time *t* the test action that minimizes the expected cost at state ξ_t should be taken, i.e.

$$A^*_{\xi_t}=\begin{cases}\forall i, 1\le i\le m; & \xi_t=[0,0,\dots 0,x]\\ \arg\min\limits_{1\le i\le m}\left\{w^*_{\xi_t}(i)\right\}; & \xi_t=\left[Y_t^{(1)},Y_t^{(2)},\dots,Y_t^{(N)},1\right]\\ \arg\min\limits_{1\le i\le m}\left\{w^*_{\xi_t}(i)+\theta^*_i v\left({\xi_{t+1}}^{(k)}\right)+\left(1-\theta^*_i\right)v\left({\xi_{t+1}}^{(0)}\right)\right\}; & \text{otherwise}\end{cases} \tag{7}$$

In general, in order to decide the optimal test action at state ξ_t, all its possible subsequent states must be considered. Also defect detection rates $\theta_i^{(k)}$ must be acquired as a priori before an optimal action can be determined.

4. Adaptive Testing Methodology

Although Proposition (6) clearly defines the optimal strategy to test the software, there are several remaining problems before it can be apply to supervise real life testing as follows:

- The set of subsequent states of ξ_t grows exponentially in proposition (6), thus the computational overhead may be unacceptable for online decision making. Also it's against common rationale to use parameters at a time point to calculate all the possible future states.
- Parameters $\theta_i^{(k)}$ must be acquired as a priori, which is not the case in real life testing, it's impossible to obtain accurate defect detection rates. The dimension of Θ is also unknown because the total number of defects is usually an unknown variable to testers.

In order to overcome these disadvantages, we developed a new adaptive testing methodology based on the original adaptive testing in [7] which adopts a recursive least square (RLSE) approach.

4.1 Parameter Estimation

There two unknown parameters that needs to be determined in proposition (6), *N* and Θ. The total number of defects *N* is required to construct the defect detection rate matrix Θ. However it might not be necessary. For example, suppose there are 10 defects in the software and we have already detected 4 of them and none of the four defects is removed and we have d = 5. Since the undetected defects are equally unknown to tester, so the estimated defect detection rates for the remaining 6 parameters should be identical, i.e., $\theta_i^{(5)}=\theta_i^{(6)}=..=\theta_i^{(10)}$. It's important to point out that this does not contradict with assumption (2) and (9) that defects are not equally detectable because these are only estimates of real defect detection rates and will be updated as testing proceeds. Note that the first detected defect locates in the first column of Θ and etc. Thus for decision making we only need to focus on column vectors $\theta_i^{(1)}$ to $\theta_i^{(5)}$ instead of the whole matrix. This means that the dimension of Θ can be limited to $m \times d$ instead of $m \times N$ which allows us to circumvent the parameter estimation problem of N which is itself a difficult problem for software engineering practitioners and scholars. Figure 4 depicts the defect detection rate matrix, whereas the dashed area is the reduced Θ. Note that parameters in the left painted region are equally unknown to tester and have identical estimates.

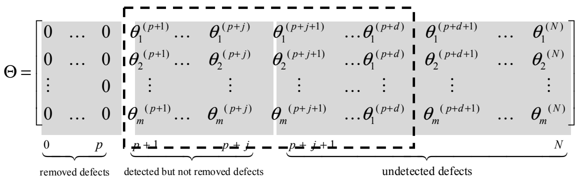

Figure 4 Defect Detection Rate Matrix

In order to obtain the estimates in Θ, a defect detection vector should be introduced, let

$$\Theta' = [\theta_1, \theta_2, \dots, \theta_m] \tag{8}$$

where θ_i denotes the probability of detecting ***any*** defect (either previous detected or undetected) by taking action *i*. And it's straight forward that there holds

$$\theta_i=\sum_{k=1}^{N}\theta_i^{(k)}, 1\le i\le m \tag{9}$$

Suppose there are *j* detected but not removed defects and *p* already removed defects, then

$$\theta_i = 0\times p+\sum_{k=p+1}^{j}\theta_i^{(k)}+(d-j)\times\theta_i^{(j+1)} \tag{10}$$

Follow the RLSE approach, Θ' can be estimated online and experimental data proved its accuracy is desirable. Now there are two set of unknown variables, $\theta_i^{(k)}(k=p+1,2,\dots,j)$ the defect detection rates for detected but not removed defects, and $\theta_i^{(j+1)}$ the detection rate for each of the undetected defects.

A natural estimation for $\theta_i^{(k)}(k=p+1,2,\dots,j)$ is

$$\theta_i^{(k)}=\frac{\text{number of detections of the } k\text{th defect}}{\text{number of test actions in class } C_i \text{ applied}} \tag{11}$$
$$(k=p+1,2,\dots,j)$$

Note that this equation does not apply for undetected defects because the numerator always equals to zero.

Finally we have the estimation algorithm for the failure detection rates of the undetected defects:

$$\hat{\theta}_i^{(j+1)} = \frac{\hat{\theta}_i - \sum_{k=p+1}^{j} \hat{\theta}_i^{(k)}}{d-p} \tag{12}$$

where $\sum_{k=p+1}^{j} \hat{\theta}_i^{(k)}$ and $\hat{\theta}_i$ denotes the estimates of $\sum_{k=p+1}^{j} \theta_i^{(k)}$ and θ_i, respectively.

The rationale behind the above algorithm is that information about detected defects which is known to the tester can be used to speculate information about the unknown defects. Also more complicated and accurate parameter estimation algorithm can be developed to improve accuracy. More details on the RLSE parameter estimation can be found in [7].

4.2 Adaptive Testing Strategy

By using the certainty-equivalence principle or the method of substituting the estimates into optimal stationary controls, we treat Θ as the true values of the corresponding parameters and determine the optimal action based on these values. Consequently, we obtain the following adaptive control policy (adaptive software testing strategy):

Step 1 Initialize parameters. Set
$\hat{\Theta} = \hat{\Theta}_0, X = M, j = 0,$ and $t = 0$.
If $M = 1$, then $A_0 = \arg \min_{1 \le i \le m} \{ w^*_{\xi_0}(i) \}$;
For other cases,
$A_0 = \arg \min_{1 \le i \le m} \{ w^*_{\xi_0}(i) + \theta^*_i v(\xi_1^{(k)}) + (1-\theta^*_i) v(\xi_1^{(0)}) \}$

Step 2 Observe the testing result by taking action A_{t+1}, i.e. whether a defect is detected.

Step 3 Follow the parameters estimation algorithm in Section 3.1 to update $\hat{\Theta}$.

Step 4 Update the current software state by setting $X = X - 1$.

Step 5 Decide the optimal action. If $X = 1$, then
$A_0 = \arg \min_{1 \le i \le m} \{ w^*_{\xi_{M-1}}(i) \}$; .
For other cases,
$A_{t+1} = \arg \min_{1 \le i \le m} \{ w^*_{\xi_t}(i) + \theta^*_i v(\xi_{t+1}^{(k)}) + (1-\theta^*_i) v(\xi_t^{(0)}) \}$

Step 6 Observe the testing result by taking action A_{t+1}, i.e. whether a defect is detected. If so set $j = j + 1$.

Step 7 If $j = d$ then take action A_{m+1} to remove the detected d defects and set
$j = j + 1$, $\theta_i^{(k)} = 0, 1 \le i \le m, 1 \le k \le d$ ·

Step 8 If $X = 0$, then stop testing; otherwise go to *Step 3*.

4.3 Reducing Computational Complexity

A possible threat to the effectiveness of the above adaptive testing strategy is that in step 5 the decision making process requires a recursive iteration to examine all possible subsequent states of the current state ξ_t. As pointed out in Section 3, the set of subsequent states of ξ_t grows exponentially, thus the computational overhead may be unacceptable for online decision making. Also it's against common sense to use parameters at one time point to predict all possible future states. So it's necessary to limit the depth of recursion to a small number. More specifically, instead of examining all possible future states till all M test actions are taken, only a number of future actions and their resulting states are examined.

Following the above analysis, Proposition (6) is replaced by the following equation

$$v(\xi_t) = \begin{cases} 0; & \text{if } X_t = 0 \\ \min_{1 \le i \le m} \{ w^*_{\xi_t}(i) \}; & \text{if } X_t < s \\ \min_{1 \le i \le m} \{ w^*_{\xi_t}(i) + \theta^*_i v(\xi_{t+1}{}^{(k)}) + (1-\theta^*_i) v(\xi_{t+1}{}^{(0)}) \}; & \\ & \text{if } X_t \ge s \end{cases} \tag{13}$$

where s denotes the number of future test actions that will be examined, i.e., the recursion depth.

5 Case Study

In this section we present a case study to examine the effectiveness of the above Adaptive Testing (AT) strategy in context of the proposed improved CMC model.

There are several variables in the improved model that affect the result of the experiments, which are: N, number of defects in the software; M, number of maximum allowed test cases; d, defect removal threshold and etc. The experiment scheme is designed to cover different combinations of these factors with different values. Two versions of the SPACE program [8] with different number of active defects are used to simulate software in different phases of testing. To compare the defect detecting (removing) performance between AT and RT, the total number of detected (removed) defect T_d in each scenario is tabulated in Table 1. Note that in order to avoid bias, each scenario is experimented 100 times and the average T_d over 100 is calculated and presented in Table . Data show that AT outperforms RT in nine out of twelve scenarios.

Table 1 Average Number of Defect Detected (removed) T_d

SUT	M=500				M=1500				M=3000			
V#1	d=1, M = 1000		d=3, M = 1000		d=1, M = 1500		d=3, M = 1500		d=1, M = 2000		d=3, M = 2000	
	34.11	33.79	**33.48**	33.32	35.13	35.32	**34.78**	34.44	**35.74**	35.72	35.24	35.6
	AT	RT	AT	RT	AT	RT	AT	RT	AT	RT	AT	RT
V#2	d=1, M = 1000		d=3, M = 1000		d=1, M = 1500		d=3, M = 1500		d=1, M = 2000		d=3, M = 2000	
	18.94	18.92	18.91	19.19	**20.24**	20.13	**20.28**	20.25	**20.84**	20.79	**20.7**	20.63
	AT	RT	AT	RT	AT	RT	AT	RT	AT	RT	AT	RT

However data in the above table shows that AT slightly outperforms RT in terms of total number of detected (removed) defect. So it's also important to investigate the testing process and defect removal process, so one of the defect removal processes is depicted in Figure 5. The horizontal axle denotes the times of defect removal. The vertical axle denotes the cumulative number of test cases consumed for each defect removal. For example, 11th point of the random curve has a value of 300, which means it takes 300 test cases for RT to begin the 11th defect removal, i.e., detects 11 x 3 = 33 defects.

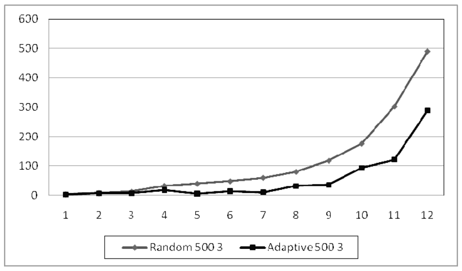

Figure 5 Defect Removal Curve

The above figure clearly depicts that AT consumes less test cases than RT to detect and remove defects throughout the testing process. The advantage increases as more defects are detected and removed.

6 Conclusions

In this paper we propose a new model for software testing that aim to reduce unrealistic assumptions adopted by previous models. Two major changes are made in the proposed model: first, we introduce batch debugging which no longer requires bugs to removed immediately upon detection; secondly, the assumption that all bugs are equally detectable is removed, i.e., each bug has its own detection rate now.

A new adaptive software testing strategy is devised and implemented to incorporate the above changes. Experimental data on the SPACE program are collected and shows that under different scenarios, AT outperforms the traditional RT in both number of detected/removed defects and the cost of the testing process.

Many future topics can be discussed, such as: more experiments on more general object programs; comparison with more advanced testing strategies; and further extend the assumptions to make it even more practical for real-life testing.

7 References

[1] K. Y. Cai, B. Gu, H. Hu, and Y. C. Li, "Adaptive software testing with fixed-memory feedback", *Journal of Systems and Software*, vol.80(8), Aug 2007, pp. 1328-1348.

[2] K. Y. Cai, C. H. Jiang, H. Hu, and C. G. Bai, "An Experimental Study of Adaptive Testing for Software Reliability Assessment", *Journal of Systems and Software*, available online, Dec 2007.

[3] K. Y. Cai, Z. Dong, K. Li and C. G. Bai, "A Mathematical Modeling Framework For Software Reliability Testing", *International Journal of General Systems*, vol.36(4), 2006, pp. 399-463(65).

[4] H. Do and G. Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," *IEEE Trans. on Software Engineering*, vol.32 (9), Sept 2006, pp.733-752.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies", *IEEE Trans. on Software Engineering*, vol. 28, Feb 2002, pp.159-182.

[6] O. Hernandez-Lerma, *Adaptive Markov Control Processes*, Springer-Verlag, 1989.

[7] H. Hu, W. E. Wong, C. H. Jiang, and K. Y. Cai, "A Case Study of the Recursive Least Squares Estimation Approach to Adaptive Testing for Software Components", *Proceedings of the Fifth international Conference on Quality Software (QSIC'05)*, Washington, DC, pp. 135-141.

[8] H. Okamura, H. Furumura and T. Dohi, "On the Effect of Fault Removal in Software Testing - Bayesian Reliability Estimation Approach", *Proceedings of the 17th International Symposium on Software Reliability Engineering ISSRE '06*, Nov 2006, pp.247-255.

[9] G. Rothermel, R. H. Untch, C. Y. Chu and M. J. Harrold, "Prioritizing test cases for regression testing", *IEEE Trans. on Software Engineering*, vol.27(10), Oct 2002, pp.929-948.