Partition Testing with Dynamic Partitioning[1]

Kai-Yuan Cai, Tao Jing and **Cheng-Gang Bai**
Department of Automatic Control
Beijing University of Aeronautics and Astronautics, Beijing 100083, China
Email: kycai@buaa.edu.cn

Abstract

A new partition testing strategy that adopts dynamic partitioning is proposed in this paper. By dynamic partitioning it is meant that the test suite or the input domain of the software under test is partitioned into several subdomains whose members are dynamically adjusted during testing by using feedback information such as the testing data collected on-line. Two controlled software experiments show that the new strategy can significantly outperform the purely random testing strategy in particular, and the random-partition testing strategy in general.

1. Introduction

Partition testing refers to a wide class of software testing techniques including statement testing, data-flow testing, branch testing, path testing, and mutation testing [1]. Existing approaches to partitioning or partition testing actually adhere to the following paradigm:

(1). The input domain of the software under test is partitioned into a number of disjoint or overlapping subdomains off-line; a test suite of a modest size is generated to achieve some direct test coverage criterion, with one or more test cases being generated for each subdomain according to a predefined test case allocation scheme; and

(2). All generated test cases are exhaustively executed against the software under test.

It has been shown that partitioning of the input domain may have significant impacts on the effectiveness of partition testing [2]. However it is not clear how to achieve an ideal partitioning. A partitioning is said to be ideal if it is convenient as well as effective. By convenient it is meant that the partitioning approach can be applied to partition the input domain and to generate the required test cases in a convenient manner without heavy overheads for partitioning. By effective it is meant that the generated test cases can effectively detect software defects such that (a). all critical defects are detected and removed from the software under test, and (b) as many other (non-critical) defects as possible are detected and removed from the software under test in a cost-effective manner.

Heavy overheads may be incurred during the off-line partitioning and test case generation. This makes existing approaches not as convenient as desired, even if the given direct test coverage criteria can be achieved in a convenient manner. The desired indirect defect coverage criteria are not guaranteed since they are not taken into account in partitioning and test case generation directly, though all generated test cases are executed. This makes existing approaches not as effective as desired. Here direct test coverage criteria refer to those criteria that can be measured directly from test cases or testing data, including statement coverage criteria, branch coverage criteria, data flow coverage criteria, functional coverage criteria, and so on. Indirect defect coverage criteria are aimed to measure how many and/or what defects can be covered or detected during testing. They are indirect in the sense that whether they are achieved can only be estimated or inferred indirectly from the test cases or testing data and cannot be measured directly since it is not clear what and how many defects are remaining in the software under test.

Following the idea of software cybernetics [3], in this paper we introduce the idea of dynamic partitioning and adhere to a different paradigm as follows

(1). A huge test suite is adopted with acceptable overheads; and

(2). The test cases are selectively executed via dynamically partitioning the test suite on-line.

Here dynamic partitioning means that the test suite is partitioned into several subdomains whose members are dynamically adjusted during testing by using feedback information such as the testing data collected on-line.

[1] Supported by the National Natural Science Foundation of China (Grant Nos: 60233020, 60474006 and 60473067).

2. The Proposed Partition Testing Strategy with Dynamic Partitioning

Suppose a huge test suite with n distinct test cases is given. Our problem is how to select test cases from the test suite and execute them against the software under test one by one.

Purely Random Testing Strategy

Step 1 No partitioning is made for the test suite;

Step 2 A test case is selected from the test suite at random in accordance with the uniform probability distribution, that is, each distinct test case has an equal probability of $1/n$ being selected;

Step 3 The selected test case is executed against the software under test and the testing results are recorded;

Step 4 If a defect is detected, then it may be removed immediately or keep un-removed for the time being;

Step 5 The executed test case is returned into the test suite and the test suite keeps invariant;

Step 6 The given testing stopping criterion is checked; if it is not satisfied, then go to Step 2;

Step 7 The software testing process is stopped.

Random-Partition Testing Strategy

Step 1 Partition the given test suite into m subdomains, which comprise $n_1, n_2, \ldots, n_m$ distinct test cases, respectively; $n_1 + n_2 + \cdots + n_m \geq n$;

Step 2 Select a subdomain in accordance with a given probability distribution $\{p_1, p_2, \ldots, p_m\}$; that is, the probability of the ith subdomain being selected is p_i; suppose the ith subdomain is selected;

Step 3 Select a test case from the ith subdomain at random in accordance with the uniform distribution; that is, each distinct test case in the ith subdomain has an equal probability of $1/n_i$ being selected;

Step 4 The selected test case is executed against the software under test and the testing results are recorded;

Step 5 If a defect is detected, then it may be removed immediately or keep un-removed for the time being;

Step 6 The executed test case is returned into the ith subdomain; the test suite and the partitioning keep invariant;

Step 7 The given testing stopping criterion is checked; if it is not satisfied, then go to Step 2;

Step 8 The software testing process is stopped.

Partition Testing Strategy with Dynamic Partitioning

Step 1 Partition the given test suite into 3 disjoint subdomains; Subdomain 0 and 2 are empty, and subdomain 1 comprises all the distinct test cases of the given test suite;

Step 2 Select a test case from subdomain 1 at random in accordance with the uniform probability distribution and execute it against the software under test;

Step 3 If the test case detects a defect, then it is moved into subdomain 2; otherwise it is moved into sudomain 0;

Step 4 If a defect is detected, then it may be removed immediately or keep un-removed for the time being;

Step 5 Check the given testing stopping criterion; if it is not satisfied, then go to Step 7;

Step 6 The software testing process is stopped;

Step 7 If subdomain 2 is not empty, then select the test case it contains and execute it against the software under test; otherwise go to Step 10;

Step 8 If the test case detects a defect, then it is kept in subdomain 2; otherwise it is moved into sudomain 1;

Step 9 Go to Step 4;

Step 10 If subdomain 1 is empty, then stop the software testing process;

Step 11 If subdomain 1 is not empty, then select a test case at random in accordance with the uniform probability distribution and execute it against the software under test;

Step 12 Go to Step 3.

Remarks

(1). The random-partition testing strategy, which was proposed by Cai et al [2], may reduce to the purely random testing strategy if each partition contains one distinct test case. In both strategies partitioning is made off-line.

(2). The partition testing strategy with dynamic partitioning (PR-DP) is an improvement over the random-partition testing strategy. Subdomains are no longer selected at random. Further, the memberships of test cases with respect to the subdomains are dynamically adjusted on-line. There is an explicit feedback mechanism adopted to improve the software testing process.

(3). The PD-DP strategy is essentially different from existing regression testing strategies [4]. First, in regression testing a test case that detects a defect is re-applied to ensure that the detected defect is removed.

After this, it is no longer purposely applied. The PD-DP strategy always re-applies the test case even if the detected defect is surely removed. The detected defects can even keep un-removed during testing. Second, some of applied test cases are often kept for regression testing even if they did not detect defects. However no explicit feedback mechanisms are adopted to select these test cases. This is not true for PD-DP. Third, partition testing and regression testing are two different forms of testing. Even in regression testing, there is a problem of input domain partitioning. Fourth, the number of classes in the PD-DP strategy can be further expanded to make a class hierarchy of test cases. Fifth, the PD-DP follows the idea of ideal partitioning.

(4). The term "dynamic partitioning" was independently adopted in the work of Chen et al [5]. However the work presented in this paper was an integral part of a comprehensive study of software reliability testing [6].

3. Experiments

3.1. Experiment I

In this experiment we applied the purely random testing strategy (PRT) and the partition testing strategy with dynamic partitioning (PR-DP) to the Space program, a subject program that was already used in the study of random testing and that of regression testing [4].

In the experiment we injected 38 defects into the Space program which was then subjected to testing. The test suite used in our case study comprised 13,498 distinct test cases, with each being an ADL file. A detected defect was removed immediately from the Space program under test. Among the 38 injected defects, two were non-detectable by any test case of the test suite. Therefore the software testing process was stopped upon the 36 detectable defects being detected. We defined such a testing process as a trial. In order to make the experiment statistically rigorous, we conducted 400 trials for software testing.

Let $b_i(k)$ be the number of tests used from the $(k-1)$th observed failure (exclusive) to the kth observed failure (inclusive). Let $t_i(k)=\sum_{j=1}^{k} b_i(j)$,

$\bar{t}(k)=\frac{1}{400}\sum_{i=1}^{400} t_i(k)$, $dn(k)=\sqrt{\frac{1}{399}\sum_{i=1}^{400}(t_i(k)-\bar{t}(k))^2}$,

$\Delta_{\bar{\Sigma}(k)}=\frac{\bar{t}(k)\text{ for PRT}-\bar{t}(k)\text{ for PT-DP}}{\bar{t}(k)\text{ for PRT}}\times 100\%$,

$\Delta_{D(k)}=\frac{dn(k)\text{ for PRT}-dn(k)\text{ for PT-DP}}{dn(k)\text{ for PRT}}\times 100\%$

Table 3.1. **Experimental results of Experiment I (%)**

k	1	2	3	4	5	6
$\Delta_{\bar{\Sigma}(k)}$	-0.24	2.23	3.04	3.22	3.60	3.61
$\Delta_{D(k)}$	2.20	16.96	19.21	26.14	28.01	28.92
k	**7**	**8**	**9**	**10**	**11**	**12**
$\Delta_{\bar{\Sigma}(k)}$	4.14	5.09	5.44	5.82	6.73	7.72
$\Delta_{D(k)}$	31.80	37.06	32.41	24.13	24.92	23.99
k	**13**	**14**	**15**	**16**	**17**	**18**
$\Delta_{\bar{\Sigma}(k)}$	9.28	10.83	11.95	12.88	14.00	14.85
$\Delta_{D(k)}$	27.12	26.56	26.51	27.73	32.92	30.45
k	**19**	**20**	**21**	**22**	**23**	**24**
$\Delta_{\bar{\Sigma}(k)}$	15.64	16.68	18.49	22.18	23.44	26.58
$\Delta_{D(k)}$	31.79	27.31	26.17	29.73	24.65	23.53
k	**25**	**26**	**27**	**28**	**29**	**30**
$\Delta_{\bar{\Sigma}(k)}$	30.50	34.80	39.59	40.87	40.97	40.69
$\Delta_{D(k)}$	24.28	32.70	38.79	32.77	30.51	29.59
k	**31**	**32**	**33**	**34**	**35**	**36**
$\Delta_{\bar{\Sigma}(k)}$	34.69	26.90	23.00	19.89	22.41	31.73
$\Delta_{D(k)}$	-3.72	-6.46	10.51	21.79	18.40	19.69

3.2. Experiment II

In this experiment the subject program was called SESD program. The SESD program was written in VC6 and comprised 3,559 lines of C++ code with 3,179 lines being executable. 28 defects were injected (with 2 being no-detectable) and the test suite comprised 5477 test cases, where a test case was simply a C language program. As in Experiment I, we conducted 400 trials for software testing.

Table 3.2. **Experimental results of Experiment II (%)**

k	1	2	3	4	5	6	7
$\Delta_{\bar{\Sigma}(k)}$	1.81	7.57	8.73	10.95	11.17	12.81	13.59
$\Delta_{D(k)}$	9.01	28.00	35.30	38.70	36.75	42.88	44.87
k	**8**	**9**	**10**	**11**	**12**	**13**	**14**
$\Delta_{\bar{\Sigma}(k)}$	15.24	15.84	16.85	17.33	17.51	17.19	16.81
$\Delta_{D(k)}$	49.09	48.22	49.32	48.32	42.45	29.48	22.18
k	**15**	**16**	**17**	**18**	**19**	**20**	**21**

$\Delta_{\overline{\Sigma}(k)}$	16.13	17.10	18.16	20.35	24.26	26.58	21.74
$\Delta_{D(k)}$	12.33	14.53	18.72	19.52	23.01	24.96	12.39
k	**22**	**23**	**24**	**25**	**26**		
$\Delta_{\overline{\Sigma}(k)}$	15.70	17.20	10.41	13.63	48.91		
$\Delta_{D(k)}$	8.92	7.63	12.62	23.30	70.02		

4. Discussion

From Table 3.1 we see that, in comparison with the purely random testing strategy, the partition testing strategy with dynamic partitioning reduced the average number of tests used for detecting the 36 defects by 31.73% and reduced the corresponding standard deviation by 19.69%. For detecting the 26 defects in Experiment II, these values became as 48.91% and 70.02%, respectively. The two experiments strongly justified the advantages of the partition testing strategy with dynamic portioning. Since the standard deviation can be viewed as a measure of stability of a testing strategy, we can reasonably say that the partition testing strategy not only requires fewer tests to detect a certain number of defects on average, but also behaves more stably than the random-partition testing strategy.

In order to answer the question concerning dynamic partitioning, we should note that dynamic partitioning is necessary as well as effective in view of defect coverage criteria. Recall the homogeneity hypothesis taken in conventional partition testing [1], which is rarely validated and does not guide partitioning in practice. In this way more than one test case should be generated for each subdomain and therefore massive test suites are helpful. Since the equivalence of two test cases for revealing failures can hardly be confirmed before being executed, in order to achieve defect coverage criteria to a greatest extent, what we can hope for is to group similar test cases into an identical subdomain and separate dissimilar test cases into different subdomains. However the testing data collected on-line may reveal that a test case dissimilar to the remaining test cases in the same subdomain is similar to the test cases in other subdomains in some sense. Also, similar test cases may no longer be similar as detected defects are removed. Dynamic partitioning becomes necessary.

On the other hand, we can justify the effectiveness or rationale of dynamic partitioning from several perspectives. First, we note that the partition testing strategy with dynamic partitioning can at least be as powerful as exhaustive selection of the test suite if software testing continues till subdomains 1 and 2 are both empty. In this case all distinct test cases are executed at least once. Second, the effectiveness of a partition is changeable during software testing and thus dynamic partitioning is required.

5. Concluding Remarks

The contribution of this paper is three-fold. First, the notion of ideal partitioning, which was largely ignored in various existing approaches to partition testing, is introduced. Second, it is shown that dynamic partitioning is necessary and valuable for a partition testing approach to be convenient as well as effective, or to be ideal. Finally, a new testing paradigm is adopted. The importance of the contribution lies not only in that it improve the effectiveness of partition testing via dynamic partitioning or on-line feedback, but also in that it further justifies the emergence of the new area of software cybernetics that explores the interplay between software and control.

6. References

[1] E.J.Weyuker, B.Jeng, " Analyzing Partition Testing Strategies" , *IEEE Transactions on Software Engineering*, Vol.17, No.7, 1991, pp703-711.

[2] K.Y.Cai, et al, " Adaptive Software Testing with Fixed-Memory Feedback", submitted for publication, 2005.

[3] K.Y.Cai, J.W.Cangussu, R.A.DeCarlo, A.P.Mathur, " An Overview of Software Cybernetics" , *Proc. the 11th International Workshop on Software Technology and Engineering Practice, 2003*, IEEE Computer Society Press, pp77-86.

[4] G.Rothermel, R.H.Untch, C.Chu, M.J.Harrold, " Prioritizing Test Cases for Regression Testing" , *IEEE Transactions on Software Engineering*, Vol.27, No.10, 2001, pp929-948.

[5] T.Y.Chen, G.Eddy, R.Merkel, P.K.Wong, " Adaptive Random Testing Through Dynamic Partitioning", *Proc. the Fourth International Conference on Quality Software*, IEEE Computer Society Press, 2004, pp70-78.

[6] T.Jing, *Reliability Wrapping, Testing, and Evaluation of Component-Based Software*, Ph.D. Dissertation, Beijing University of Aeronautics and Astronautics, 2004.