2009

Improving software testing cost-effectiveness through dynamic partitioning

Zhiquan Zhou
University of Wollongong, zhiquan@uow.edu.au

Arnaldo Sinaga
University of Wollongong, ams939@uowmail.edu.au

Lei Zhao
Beijing University of Aeronautics and Astronautics

Willy Susilo
University of Wollongong, wsusilo@uow.edu.au

Kai-Yuan Cai
Beijing University of Aeronautics and Astronautics

Improving software testing cost-effectiveness through dynamic partitioning

Abstract

We present a dynamic partitioning strategy that selects test cases using online feedback information. The presented strategy differs from conventional approaches. Firstly, the partitioning is carried out online rather than off-line. Secondly, the partitioning is not based on program code or specifications; instead, it is simply based on the fail or pass information of previously executed test cases and, hence, can be implemented in the absence of the source code or specification of the program under test. The cost-effectiveness of the proposed strategy has been empirically investigated with three programs, namely SPACE, SED, and GREP. The results show that the proposed strategy achieves a significant saving in terms of total number of test cases executed to detect all faults.

Disciplines

Physical Sciences and Mathematics

Publication Details

Sinaga, A., Zhou, Z., Susilo, W., Zhao, L. & Cai, K. 2009, "Improving software testing cost-effectiveness through dynamic partitioning", in B. Choi (eds), Proceedings of the 9th International Conference on Quality Software, IEEE, Los Alamitos, USA, pp. 249-258.

2009 Ninth International Conference on Quality Software

Improving Software Testing Cost-Effectiveness Through Dynamic Partitioning

Zhi Quan Zhou*, Arnaldo Sinaga*, Lei Zhao†, Willy Susilo* and Kai-Yuan Cai†
*School of Computer Science and Software Engineering
University of Wollongong
Wollongong, NSW 2522, Australia
Email: {zhiquan, ams939, wsusilo}@uow.edu.au
† Department of Automatic Control
Beijing University of Aeronautics and Astronautics
Beijing 100191, China
Email: zhaolei@asee.buaa.edu.cn, kycai@buaa.edu.cn

***Abstract*—We present a dynamic partitioning strategy that selects test cases using online feedback information. The presented strategy differs from conventional approaches. Firstly, the partitioning is carried out online rather than off-line. Secondly, the partitioning is not based on program code or specifications; instead, it is simply based on the fail or pass information of previously executed test cases and, hence, can be implemented in the absence of the source code or specification of the program under test. The cost-effectiveness of the proposed strategy has been empirically investigated with three programs, namely SPACE, SED, and GREP. The results show that the proposed strategy achieves a significant saving in terms of total number of test cases executed to detect all faults.**

***Keywords*-software testing; random testing; partition testing; dynamic partitioning.**

I. Introduction

It is widely accepted that the cost of testing, debugging, and verification activities can easily range from 50 to 75 percent of the total development cost in a typical commercial development organization [1]. Among the various software testing methods, *random testing* [2] is the most fundamental. It selects test cases randomly, hence avoiding the overhead of program- or specification-based partitioning of the input domain. Random testing has often been employed to test real-world applications (for example, [3], [4]).

In random testing, each test case is selected independently. To make the detection of the first failure quicker (that is, to reduce the number of test cases needed to detect the first failure), Malaiya introduced an *antirandom testing* technique [5], where the first test case is selected randomly, and each subsequent test case is selected by choosing the one whose total distance to all the previously executed test cases is maximum. In antirandom testing, the total number of test cases has to be decided in the first place. The randomness of this method is also very limited because only the first test case is selected randomly; the sequence of all the subsequent test cases is deterministic.

More recently, another testing strategy named *Adaptive Random Testing* (ART) has been proposed to improve the fault-detection capability of random testing without the above limitations [6], [7], [8], [9]. ART is based on the intuition that when failure-causing inputs are clustered, selecting an input close to the previously executed test cases that have not revealed a failure is less likely to detect a failure. ART, therefore, proposes to have test cases evenly spread throughout the input domain. It differs from antirandom testing in that it well preserves the randomness since all test cases in ART are randomly selected, and that ART does not require the predetermination of the total number of test cases.

ART is developed as an enhancement to random testing with an objective of using fewer test cases to detect the first failure. A related technique, known as *adaptive testing*, has been developed by Cai et al. [10] with a different objective and approach. Adaptive testing adjusts the selections of test actions online following the idea of adaptive control, to achieve an optimization objective, such as minimizing the total cost of detecting and removing multiple faults. Adaptive testing involves partitioning of the input domain. The partitioning is made off-line. It has been shown that partitioning strategies can have a significant impact on the total cost of testing [10].

The question of how to achieve effective partitioning without heavy overheads is further investigated by Cai et al. in [11]. Following the idea of *software cybernetics* [12], Cai et al. proposed a dynamic partitioning approach by adopting a new testing paradigm as follows: Firstly, a huge test suite is given; secondly, test cases are selectively executed through dynamically partitioning the test suite online. In general, software cybernetics explores the interplay between software

All correspondence should be addressed to Dr. Zhi Quan Zhou, School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW 2522, Australia. Email: zhiquan@uow.edu.au. Telephone: (61-2) 4221 5399.

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.40

and control.

As this paper extends the work of Cai et al. [11], we would like to briefly introduce the dynamic partitioning strategy proposed in [11]. The strategy is based on the intuition that a test case that has detected a fault previously should have a higher fault-detection capability than those that have not detected a fault previously. The strategy, therefore, partitions the given test suite into three categories. Initially, all test cases are in category 1, from which a test case t will be randomly chosen to run. If t does not reveal a failure, then t is considered less powerful, and is moved from category 1 to category 0; otherwise t is considered to be powerful, and is moved from category 1 to category 2. After a failure is detected, the program under test may be modified to remove the fault. It should be noted that, in practice, such an attempt of modifying programs may not always correctly remove a fault, and sometimes may introduce new faults. In any case, the program needs to be tested again. Now the test case t in category 2 will be selected again to test the modified version of the program. If, at this time, no failure is detected, then t will be moved back to category 1; otherwise it will remain in category 2 to be used again next time. If category 2 is empty, then a test case will be selected from category 1 randomly. If category 1 is also empty, the testing process will terminate. Therefore, this strategy partitions the given test suite online into three disjoint categories, and gives priority to category 2, followed by category 1. The membership of each test case changes dynamically. Cai et al. experimentally compared this dynamic partitioning strategy with two other random testing strategies and showed that the dynamic partitioning strategy outperformed the other two [11].

We note that the dynamic partitioning strategy proposed in [11] is relatively simple: The category 2 always contains no more than one test case; when the test case in category 2 no longer detects a failure, it is simply moved back to category 1; when both categories 1 and 2 become empty, the testing process terminates. Furthermore, in all the test case selection strategies (including random testing) investigated in [11], the selection of test cases is by sampling *with* replacement. In real-world software testing, it is more practical to use sampling *without* replacement to save cost.

This paper extends the work of Cai et al. [11] by developing enhanced versions of dynamic partitioning strategies in the context of sampling without replacement, and providing further empirical study results to show the cost-effectiveness of dynamic partitioning strategies.

II. The Test Case Selection Algorithms

A. Objectives

In real-world large-scale software development, the program under test normally contains multiple faults in its initial version, and will undergo many rounds of testing, debugging, and retesting cycles. Let $T = \{t_1, t_2, \ldots, t_n\}$ be a test suite with n distinct test cases, and n is very large. T is created according to certain criteria. The program needs to be tested by selectively executing test cases from T. Suppose a failure is detected after a certain number of test cases from T have been run. Then an attempt is made to remove the fault, yielding a new version. The question is how we should test this new version (and its subsequent versions)? Ideally, we should first apply a regression testing technique [13] to rerun some or all of the previously executed test cases; if no failure is detected, then we should continue with test cases that have not been applied before. It must be pointed out, however, that real-world software testing is always carried out with limited resources, and an important contributor to the cost of testing is the total number of test cases to execute. Suppose we can only afford the executions of m test cases in total for the entire testing process, then what is the most cost-effective strategy for selecting test cases for each version of the program under test? That is, how should we select test cases, which total to m for all versions, so that they can detect as many faults as possible? Or, if the stopping criterion is to release the program after r faults have been removed, then how can we run as few test cases as possible to achieve this goal? For practical purposes, we also require that the test case selection strategy be easy to adopt without heavy overheads or the need of sophisticated tool support. Note that the context and objectives stated above are very different from those of regression testing.

In the following subsections, four test case selection algorithms will be proposed to investigate the above question.

B. Pure Random Testing (PRT)

This subsection proposes a Pure Random Testing (PRT) algorithm. Figure 1 shows both the algorithm and how its cost-effectiveness is measured. This algorithm will serve as a benchmark in our empirical study.

In Figure 1 , words enclosed between "/*" and "*/" are comments. Line 1 of the algorithm initializes two variables, which will be used as counters to record the up-to-date number of failures detected and number of test cases executed. These two numbers indicate the cost-effectiveness of testing. A cost-effective testing algorithm should use a small number of test cases to detect a large number of failures.

Lines 2 and 3 initialize two sets of test cases, namely Set_0 and Set_1. Set_0 stores test cases that have already been run for the *current* version of the program under test, whereas Set_1 stores test cases that have not been run for the current version. Initially, Set_0 is empty and Set_1 contains all the test cases.

Line 4 controls when the algorithm should terminate. When *either* of the following two conditions is met, the algorithm will terminate. Condition 1: Set_1 becomes empty, which means that the current version of the program under test has passed all the given test cases. (It is beyond the scope of this paper to discuss how to continue testing by generating new test cases from outside of the given test suite.

Purpose: This is a Pure Random Testing algorithm, which tests a given program *P* using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (*n* is very large). *P* can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize Set0 to empty. /* to store used test cases */
3.   Initialize Set1 to {t1, t2, …, tn}. /* to store unused test cases */
4.   While (Set1 is not empty and the given testing stopping criterion is not met)
5.      Randomly select a test case ti from Set1.
6.      Test P using ti.
7.      Increase nbOfAllTests by 1. /* record the number of executed test cases */
8.      If (no failure is detected)
        Then
9.          Move ti from Set1 to Set0 /* so ti will not be chosen again */
        Else
10.         Increase nbOfDetectedFailures by 1.
11.         Print nbOfDetectedFailures and nbOfAllTests. /* current cost-effectiveness */
12.         An attempt can be made to remove the fault from P.
13.         Move all elements in Set0 to Set1. /* Hence, Set0 will become empty. */
        EndIf
14.  EndWhile
15.  Print nbOfDetectedFailures and nbOfAllTests. /* overall cost-effectiveness */
     End of Algorithm
```

Figure 1. The Pure Random Testing (PRT) Algorithm

Purpose: This is a Regression-Random Testing algorithm, which tests a given program *P* using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (*n* is very large). *P* can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize Set0 to empty.
3.   Initialize Set1 to {t1, t2, …, tn}.
4.   Initialize currentSet to empty. /* currentSet always contains no more than one element */
5.   Randomly select a test case ti from Set1, and move ti from Set1 to currentSet.
6.   While (currentSet is not empty and the given testing stopping criterion is not met)
7.      Test P using the test case in currentSet.
8.      Increase nbOfAllTests by 1.
9.      If (no failure is detected)
        Then
10.         Move the test case in currentSet to Set0.
11.         If Set1 is not empty, then randomly select a test case from Set1, and move
            it to currentSet.
        Else
12.         Increase nbOfDetectedFailures by 1.
13.         Print nbOfDetectedFailures and nbOfAllTests.
14.         An attempt can be made to remove the fault from P.
15.         Move all elements in Set0 to Set1. /* The element in currentSet remains there. */
        EndIf
16.  EndWhile
17.  Print nbOfDetectedFailures and nbOfAllTests.
     End of Algorithm
```

Figure 2. The Regression-Random Testing (RRT) Algorithm

The given test suite is assumed to be huge and, in fact, it can even refer to the entire input domain.) Condition 2: The given testing stopping criterion is met. Such a criterion can be, for instance, a prescribed number of faults have been detected, or a prescribed number of test cases have been run.

In lines 5 and 6, a test case is randomly selected from Set_1 and run. Line 7 updates the counter to record the number of test cases executed so far. Line 8 checks whether a failure is detected. If no failure is detected, the control goes to line 9, which deletes the current test case from Set_1 and puts it into Set_0. Set_0 stores test cases already executed for the current version. Members of Set_0 will never be selected to test the program. This strategy of *sampling without replacement* is used in all the algorithms proposed in this paper.

If a failure is detected, then the control goes to line 10, which increases the counter to record the number of failures detected so far. The values of this counter and the other counter (see line 7) serve as the up-to-date cost-effectiveness indicators and are printed in line 11. Since a failure is detected, an attempt can be made to remove the fault from the program under test, as indicated in line 12. In practice, however, such an attempt to modify programs may not necessarily remove the genuine fault, and may sometimes introduce new faults. Hence, the new version of the program needs to be tested on all the test cases in Set_0 and Set_1. Therefore, in line 13 all the test cases in Set_0 are moved back to Set_1 so that they can be selected again.

Before the algorithm terminates, the overall cost-effectiveness is printed in line 15.

C. Regression-Random Testing (RRT)

In PRT, the test case that has just detected a failure is treated in the same way as all the other test cases. This subsection proposes an algorithm that gives higher priority to the last failure-causing input: Once a failure is detected, the failure-causing input will be repeatedly applied until no more failure can be detected. This algorithm is based on the intuition that a test case that has just detected a failure might be able to detect another failure in the next test. The algorithm is also based on the common practice that after the program under test is modified with an attempt to remove a fault, the modified version is often first retested using the last failure-causing input. This is a simple regression testing strategy.

The algorithm is shown in Figure 2 . Lines 1 to 3 are the same as those of the PRT algorithm. Line 4 initializes an empty set $currentSet$. This set always contains no more than one element, namely the test case currently being executed. Line 5 randomly selects a test case from the test suite and put it into $currentSet$. The two termination conditions checked in line 6 are similar to those of the PRT algorithm except that it is $currentSet$ instead of Set_1 that is checked. The logic of lines 7 to 10 are similar to that of the PRT algorithm. Line 11 randomly selects a new test case from Set_1 (if Set_1 is empty, then no test case will be selected) and move this test case to $currentSet$. The logic of lines 12 to 15 is similar to that of the PRT algorithm. Note, however, that in this branch (lines 12 to 15) no test case is selected from Set_1 and, therefore, the present element in Set_1 will be used again next time since it has just detected a failure.

The algorithm is named Regression-Random Testing (RRT), where "Regression" refers to the phase that reruns the last failure-causing input, and "Random" refers to the phase that randomly selects a test case from the set of all unused test cases when the regression testing phase cannot detect a failure.

As we do not assume the availability of sophisticated change-tracking tools, no further regression testing techniques will be incorporated.

D. Testing Through Dynamic Partitioning with Fixed Membership (DPFM)

In RRT, once the previous failure-causing input no longer detects a failure, it will be treated as an ordinary test case and will not be given priority any more in future test case selection. It is, however, an intuition that test cases that have detected failures in the past may be powerful in detecting a failure again in the future. Based on this intuition, two test case selection algorithms will be proposed in this and the next subsections.

The algorithm shown in Figure 3 is named Dynamic Partitioning with Fixed Membership (DPFM) algorithm. It partitions the given test suite into four disjoint sets, namely $fair$, $good$, $poor$, and $currentSet$. The set $currentSet$ is used in the same way as explained in the RRT algorithm: It stores the test case currently being executed and, hence, always contains no more than one element. For the other three sets ($fair$, $good$, and $poor$), each of them is further divided into two parts, namely the *used* part, which stores test cases that have already been applied for the current version of the program P, and the *unused* part, which stores test cases not yet applied for the current version of P. Initially, all test cases are stored in $fair\_unused$ (that is, the unused part of the $fair$ set), as indicated in line 2 of the algorithm. All the other sets are initialized to be empty, as indicated in line 3.

The algorithm works as follows. Initially, a test case, say t_i, is randomly selected from the $fair$ set. If no failure is detected, t_i will be considered to be a less powerful test case and moved to the $poor$ set; otherwise t_i will be repeatedly applied to test P, just in the same way as the RRT algorithm, until no more failure can be detected, and then t_i will be moved to the $good$ set because it is considered to be a powerful test case. Once t_i is moved to either the $poor$ or the $good$ set, the membership of t_i will never be changed (hence, the algorithm is named "Fixed Membership"). Every

Purpose: This is an algorithm for testing through Dynamic Partitioning with Fixed Membership. It tests a given program *P* using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (*n* is very large). *P* can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize fair_unused to {t1, t2, …, tn}.
3.   Initialize good_unused, good_used, fair_used, poor_unused, poor_used,
     and currentSet to empty.
4.   Initialize failureDetected to false.
5.   Initialize currentSetName to "fair".
6.   Randomly select a test case ti from fair_unused, and move ti to currentSet.
7.   While (currentSet is not empty and the given testing stopping criterion is not met)
8.      Test P using the test case in currentSet.
9.      Increase nbOfAllTests by 1.
10.     If (no failure is detected)
11.     Then Call sub-algorithm removeFromCurrentSet.
12.          Call sub-algorithm moveToCurrentSet.
13.          Set failureDetected to false.
14.     Else Increase nbOfDetectedFailures by 1.
15.          Print nbOfDetectedFailures and nbOfAllTests.
16.          An attempt can be made to remove the fault from P.
17.          Move all elements in good_used to good_unused, in fair_used to fair_unused, and
             in poor_used to poor_unused.
18.          Set failureDetected to true.
        EndIf
19.  EndWhile
20.  Print nbOfDetectedFailures and nbOfAllTests.
     End of Algorithm

     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "good")
2.   Then Move the test case in currentSet to good_used.
3.   Else If (currentSetName is "fair")
          Then
4.            If (failureDetected is false)
5.            Then Move the test case in currentSet to poor_used.
6.            Else Move the test case in currentSet to good_used.
              EndIf
          Else /* currentSetName is "poor" */
7.            Move the test case in currentSet to poor_used.
          EndIf
     EndIf
     End of Sub-Algorithm removeFromCurrentSet

     Begin Sub-Algorithm moveToCurrentSet
1.   If (good_unused is not empty)
2.   Then Randomly select a test case ti from good_unused.
3.        Move ti to currentSet.
4.        Set currentSetName to "good".
5.   Else If (fair_unused is not empty)
6.        Then Randomly select a test case ti from fair_unused.
7.             Move ti to currentSet.
8.             Set currentSetName to "fair".
9.        Else If (poor_unused is not empty)
               Then
10.                Randomly select a test case ti from poor_unused.
11.                Move ti to currentSet.
12.                Set currentSetName to "poor".
               EndIf
          EndIf
     EndIf
     End of Sub-Algorithm moveToCurrentSet
```

Figure 3. The Dynamic Partitioning with Fixed Membership (DPFM) Algorithm

time a new test case is needed, the algorithm will first look at the unused part of the $good$ set; if it is empty, then the $fair$ set; if it is also empty, then the $poor$ set.

Let us elaborate on the above procedure. Lines 4 and 5 initialize two variables, where $failureDetected$ is a flag indicating whether a failure has been detected, and $currentSetName$ records where the element of $currentSet$ has been selected from. It hence can only take three values, namely "$good$", "$fair$", or "$poor$". Line 6 randomly selects a test case from the given test suite, and moves it to $currentSet$. The program P is then tested in line 8 using the selected test case. If a failure is detected, the control goes to line 14. The logic from line 14 to line 17 is very similar to that of the RRT algorithm. Line 18 sets the flag to $true$ to indicate that a failure has been detected. Note that $currentSet$ has not been updated and, hence, the same test case will be used again to test the program P in the next iteration.

If no failure is detected, the control will go to line 11 to call the sub-algorithm *removeFromCurrentSet*. This sub-algorithm moves the element in $currentSet$ to one of the other sets as follows: If the element was selected from the $good$ or $poor$ set, it will be returned to the same set because of the use of fixed membership. If the element was selected from the $fair$ set, it will be moved to the $good$ set if it detected a failure last time; otherwise it will be moved to the $poor$ set. After this sub-algorithm returns, $currentSet$ will become empty. Then line 12 calls another sub-algorithm *moveToCurrentSet*, which selects a new test case (with the $good$ set having the highest priority, followed by the $fair$ set, and the $poor$ set has the lowest priority) and moves it to $currentSet$. Note that if the unused parts of all the three sets are empty, then $currentSet$ will remain empty, which will cause the loop to terminate. Finally, line 13 sets the flag to record the up-to-date test result.

E. Testing Through Dynamic Partitioning with One-Step Varying Membership (DP1S)

In the DPFM algorithm, if a test case selected from the $good$ set no longer detects a failure, it is still returned to the $good$ set. On the other hand, test cases selected from the $poor$ set will always be returned to the $poor$ set regardless of whether they can detect a failure or not. This strategy does not reflect the fact that in the real world a "good" test case might become "poor", and a "poor" test case might become "good". This subsection, therefore, proposes an algorithm that can change the membership of test cases in real time.

The algorithm is shown in Figure 4 . It differs from the DPFM algorithm only in the sub-algorithm *removeFromCurrentSet*. Let t_i be the test case in $currentSet$. When t_i no longer detects a failure, it will be removed from $currentSet$ as follows. If t_i was selected from the $good$ set, it will be returned to the $good$ set only if it detected a failure in the last test; otherwise it will be degraded by one step to the $fair$ set (see lines 3 and 4). If t_i was selected from the $fair$ set, it will be either upgraded or degraded by one step depending on whether a failure was detected in the last test (see lines 7 and 8). If t_i was selected from the $poor$ set, it will be returned to the $poor$ set if no failure was detected; otherwise it will be upgraded by one step to the $fair$ set (see lines 10 and 11). The algorithm is therefore named *Testing Through Dynamic Partitioning with One-Step Varying Membership* (DP1S), where "One-Step Varying Membership" refers to the strategy that upgrades or degrades by one step every time the membership is to be changed.

III. The Experiments

To empirically investigate the cost-effectiveness of the algorithms proposed in Section II , they were applied to test three real-world programs, namely the SPACE, SED, and GREP programs. These programs were downloaded from the *Software-artifact Infrastructure Repository* (http://sir.unl.edu) [14], which is a repository of subject programs that provide a common ground for comparing different testing techniques. Each package of subject program provides both the original and associated faulty versions as well as a suite of test cases. For each subject program, we combined the faults into one version to create a program that contains multiple faults. During an execution, if any fault is encountered, its label will be recorded in the log file. The original version was used as the test oracle. Every time a test case is executed, the output of the faulty version is compared with that of the original version, and a discrepancy indicates a failure. When a failure is detected, the log file will be checked, and the first fault encountered in the execution will be removed to simulate the debugging process. To "remove a fault", we delete the corresponding faulty statement(s) and restore the corresponding original statement(s). While a fault thus removed might not be the genuine cause for the failure, this process faithfully simulates the debugging process because when the debugger manually traces the failed execution, the first fault in the execution path will have a higher chance to be identified and corrected first. The testing stopping criterion is that when all the seeded faults have been removed, the testing will stop. The total number of test cases executed will indicate the overall cost-effectiveness: The smaller, the better. For each algorithm and each program under test, the experiment has been repeated one hundred times.

A. Results of Experiments with SPACE

SPACE is a subject program that has often been used in the study of testing effectiveness [15]. It consists of 6,199 lines of (executable) C code and 136 functions, and works as an interpreter for an array definition language. The faulty version used in our experiment involves 33 faults which, according to the Software-artifact Infrastructure Repository,

```
Note: This is an algorithm for testing through Dynamic Partitioning with One-Step Varying
Membership. This algorithm differs from the Dynamic Partitioning with Fixed Membership (DPFM)
algorithm only in the following sub-algorithm.

     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "good")
2.   Then If (failureDetected is false)
3.        Then Move the test case in currentSet to fair_used.
4.        Else Move the test case in currentSet to good_used.
          EndIf
5.   Else If (currentSetName is "fair")
          Then
6.            If (failureDetected is false)
7.            Then Move the test case in currentSet to poor_used.
8.            Else Move the test case in currentSet to good_used.
              EndIf
          Else /* currentSetName is "poor" */
9.            If (failureDetected is false)
10.           Then Move the test case in currentSet to poor_used.
11.           Else Move the test case in currentSet to fair_used.
              EndIf
          EndIf
     EndIf
     End of Sub-Algorithm removeFromCurrentSet
```

Figure 4. The Dynamic Partitioning with One-Step Varying Membership (DP1S) Algorithm

Table I
RESULTS OF EXPERIMENTS WITH SPACE (33 FAULTS, 13,551 TEST CASES, 100 TRIALS)

Algorithm	Average	Median	Std Deviation
PRT	2390	2117	1161
RRT	1699	1308	1272
DPFM	1861	1537	1345
DP1S	1608	1307	1065

were real faults discovered during the program's development. The 13,551 test cases included in the package have been used in our experiments.

Table I summarizes the experimental results. To detect and remove all the 33 faults seeded in the SPACE program, the Pure Random Testing (PRT) algorithm executed a total of 2,390 test cases in average. As a comparison, the RRT algorithm used only 1,699 test cases. In other words, it achieved a saving of 28.9% in average. [1] The average cost-effectiveness of the DPFM algorithm lies in between PRT and RRT, with a saving of 22.1%. The DP1S algorithm has been the best among all the four, with an average saving of 32.7%. A similar comparison can be made for medians. It is interesting to note, however, that the standard deviations of these algorithms followed a different ranking: While DP1S is still the best, the PRT algorithm demonstrated a higher stability than the RRT and DPFM algorithms.

In conclusion, all the algorithms that employ online feedback information (that is, RRT, DPFM, and DP1S) outperformed the Pure Random Testing algorithm in cost-effectiveness, and the DP1S algorithm further outperformed all the other algorithms in both the cost-effectiveness and the stability.

Figure 5 depicts the average results for detecting each failure. As there are 33 faults seeded into the program, and each fault-removal activity removes one and only one fault, a total of 33 failures will have been detected when the last fault has been removed. Figure 5 shows that until the detection of the 20th failure or so, the number of executed test cases has been quite small for all algorithms (ranging from 40 to 50). The numbers of test cases increase significantly in the later stage of testing, from failure 28 or so. This is reasonable because, when there are many faults in the program, the failure rate is high and, hence, it is easy to detect a failure even with Pure Random Testing. When more and more faults are removed from the program, the failure rate gets smaller (in other words, it becomes more expensive to detect a failure), and the algorithms employing feedback information achieve considerable savings.

B. Results of Experiments with SED

SED (http://www.gnu.org/software/sed) is a stream editor. It is a Unix/Linux utility and performs text transformations on an input stream. The program used in our experiments is written in C, consisting of 14,427 lines of code and 255 functions. The downloaded package includes seven version pairs of the SED program (from version 1 to version 7), where each pair consists of an original version and its corresponding faulty version with seeded faults. The latest version, namely version 7, was used in

[1] In this paper, any "saving" refers to the comparison with the Pure Random Testing algorithm.

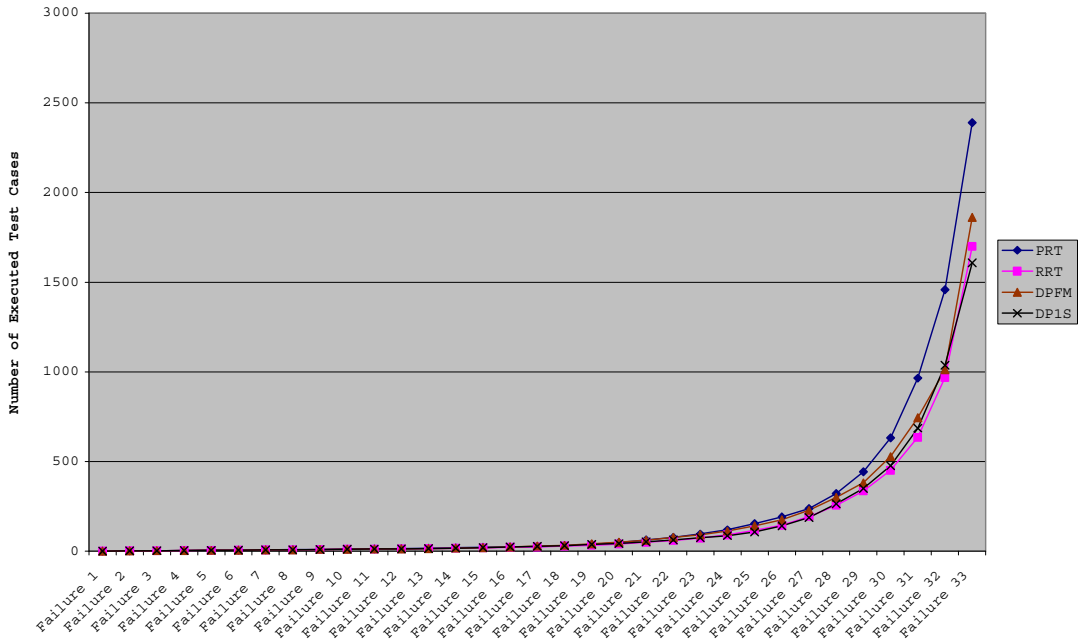

Figure 5. Average Numbers of Test Cases Executed to Detect Each Failure (for the SPACE Program)

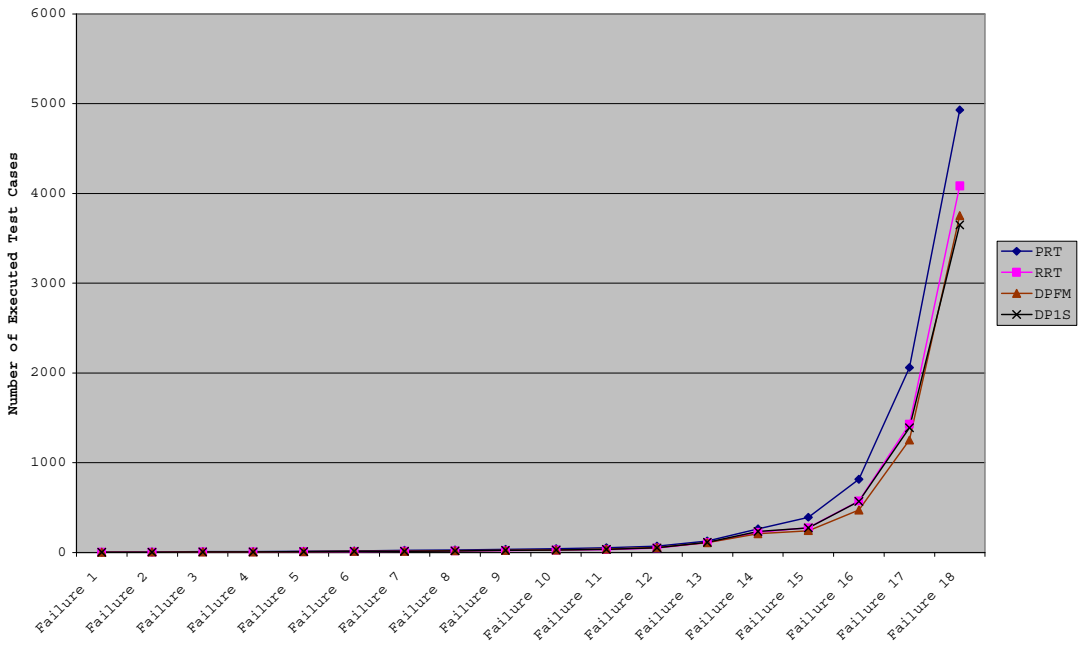

Figure 6. Average Numbers of Test Cases Executed to Detect Each Failure (for the SED Program)

our experiments. A total of 18 faults were injected to create a version that involves multiple faults. These 18 faults were taken not only from the faulty version 7, but also from the faulty versions 5 and 6. Our test suite consists of 12,238 test cases, of which 415 were provided directly by the downloaded package, and the other 11,823 were generated using the additional input data (namely SED scripts and input files) provided by the downloaded package, and some of the SED scripts were also taken from the SED manual available in the GNU Web site.

Table II
RESULTS OF EXPERIMENTS WITH SED (18 FAULTS, 12,238 TEST CASES, 100 TRIALS)

Algorithm	Average	Median	Std Deviation
PRT	4932	4472	2727
RRT	4085	3579	2407
DPFM	3755	3568	1970
DP1S	3650	3316	2032

Table III
RESULTS OF EXPERIMENTS WITH GREP (22 FAULTS, 10,065 TEST CASES, 100 TRIALS)

Algorithm	Average	Median	Std Deviation
PRT	1152	820	996
RRT	1113	915	943
DPFM	928	704	767
DP1S	903	711	759

The experimental results are summarized in Table II . To detect and remove all the 18 faults, the PRT algorithm used a total of 4,932 test cases in average. The RRT, DPFM, and DP1S algorithms achieved a saving of 17.2%, 23.9%, and 26.0%, respectively. A similar comparison can be made for medians. Obviously, the DP1S algorithm was still the best in cost-effectiveness. The DPFM and DP1S algorithms also outperformed the PRT and RRT algorithms in stability (standard deviations).

The average results for detecting each failure are shown in Figure 6 . It can be observed that the number of test cases needed to detect a failure in the later stage of testing increases dramatically. The algorithms employing feedback information, therefore, bring significant savings.

C. Results of Experiments with GREP

GREP (http://www.gnu.org/software/grep) is another Unix/Linux utility. It searches input files for lines containing a match to a specified pattern. The program contains 10,068 lines of C code and 146 functions. Using an approach similar to that for the SED program, we seeded 22 faults into the program and created a suite of 10,065 test cases. Experimental results are shown in Table III . To detect and remove all the 22 faults, PRT used a total of 1,152 test cases in average. The RRT, DPFM, and DP1S algorithms achieved a saving of 3.4%, 19.4%, and 21.6%, respectively. DPFM and DP1S also outperformed the other two in terms of medians, and the DP1S algorithm was the most stable in terms of standard deviation. The average results for detecting each failure are shown in Figure 7 .

IV. CONCLUSION

This paper considers test case selection in a dynamic process, where the software under test may change over time. Our techniques also relate to test case prioritization [16]. Four algorithms have been proposed, where PRT serves as a benchmark, and the other three employ online feedback information to dynamically partition the given test suite. An advantage of these algorithms is that they are easy to implement without the need of advanced supporting tools, such as change tracking tools, or the source code or the specification of the program under test. The cost-effectiveness of these algorithms has been empirically investigated in terms of the total number of test cases needed to detect and remove all the faults seeded into the program under test. Experimental results show that all the three algorithms employing feedback information yielded savings in average compared with the PRT algorithm, and that DP1S has been the most cost-effective and most stable algorithm with an average saving of 32.7% for SPACE, 26.0% for SED, and 21.6% for GREP. This magnitude of saving can significantly reduce the cost of real-world large-scale software development, maintenance, and evolution, where the total number of test cases can be very high. This result agrees with our expectation because DP1S employs more feedback information than the other algorithms: PRT does not employ any feedback information; RRT employs limited feedback information, which is only about the last (one) test case; DPFM does not utilize the up-to-date performance information of test cases once their membership is fixed. DP1S, on the other hand, adjusts the membership of test cases dynamically and gradually based on their up-to-date performance.

We have not compared the results reported in this paper with those reported in [11]. This is because they are not directly comparable as the former uses sampling without replacement whereas the latter uses sampling with replacement. The results of this research further justify the emergence of the area of software cybernetics. In future research we will enhance the algorithms by incorporating other types of feedback information.

ACKNOWLEDGMENTS

Zhou is supported in part by a Small Grant of the University of Wollongong. Cai is supported in part by the National Science Foundation of China and Microsoft Research Asia (Grant No. 60633010). We would like to thank Ke Cai for his discussions on the design of the algorithms and helps with the implementation.

REFERENCES

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] P. S. Loo and W. K. Tsai, "Random testing revisited," *Information and Software Technology*, vol. 30, no. 7, pp. 402–417, 1988.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

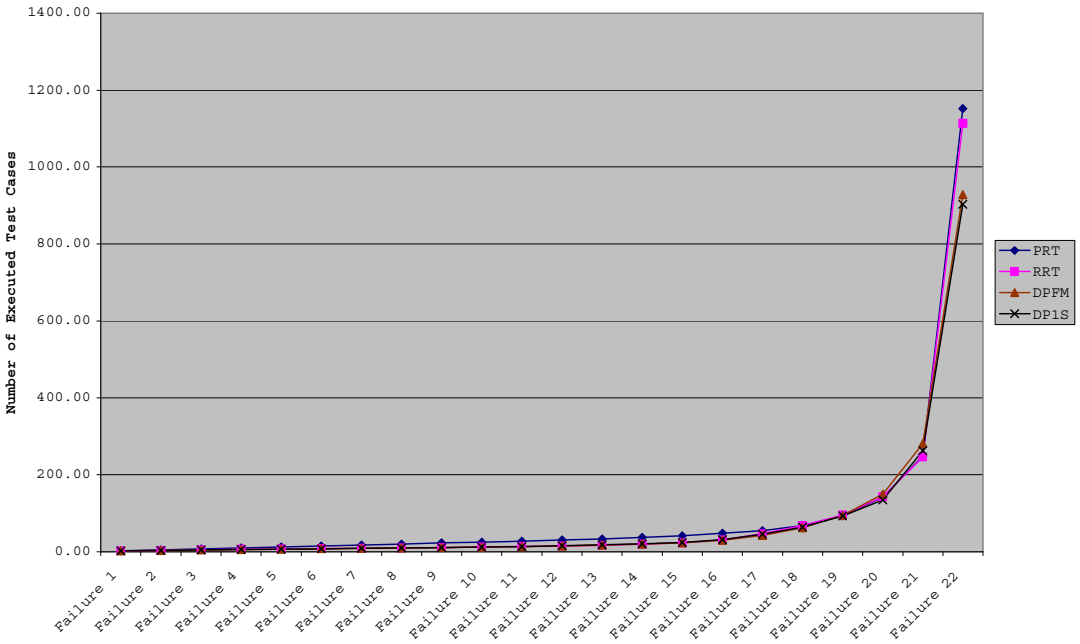

Figure 7. Average Numbers of Test Cases Executed to Detect Each Failure (for the GREP Program)

[4] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for java JIT compiler test system," in *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)*. IEEE Computer Society Press, 2003, pp. 20–24.

[5] Y. K. Malaiya, "Antirandom testing: Getting the most out of black-box testing," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, 1995, pp. 86–95.

[6] I. K. Mak, "On the effectiveness of random testing," Master's thesis, The University of Melbourne, Melbourne, Australia, 1997.

[7] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), Lecture Notes in Computer Science 3321*. Springer-Verlag, 2004, pp. 320–329.

[8] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "On favourable conditions for adaptive random testing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 6, pp. 805–825, 2007.

[9] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software*, to appear.

[10] 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, pp. 1328–1348, 2007.

[11] K.-Y. Cai, T. Jing, and C.-G. Bai, "Partition testing with dynamic partitioning," in *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*. IEEE Computer Society Press, 2005, pp. 113–116.

[12] K.-Y. Cai, T. Y. Chen, and T. H. Tse, "Towards research on software cybernetics," in *Proceedings of 7th IEEE International Symposium on High Assurance Systems Engineering*. IEEE Computer Society Press, 2002, pp. 240–241.

[13] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, 2001.

[14] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[15] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 44–53.

[16] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *Proceedings of the World Wide Web Conference*. ACM Press, 2009, pp. 901–910.