Bi-criteria Test Suite Reduction by Cluster Analysis of Execution Profiles

Alireza Khalilian, Saeed Parsa

Bi-Criteria Test Suite Reduction by Cluster Analysis of Execution Profiles

Alireza Khalilian and Saeed Parsa

Iran University of Science and Technology, Tehran, Iran
khalilian@comp.iust.ac.ir, parsa@iust.ac.ir

Abstract. The aim has been to minimize regression test suites while retaining fault detection capability of the test suite admissible. An appropriate minimized test suite should exercise different execution paths within a program. However, minimization of test suites may result in significant fault detection loss. To alleviate the loss, a new bi-criteria heuristic algorithm, using cluster analysis of test cases execution profiles is proposed in this paper. Cluster analysis of execution profiles categorizes test cases according to their similarity in terms of exercising a certain coverage criterion. Considering additional coverage criteria the proposed algorithm samples some test cases from each cluster. These additional criteria exercise execution paths, different from those covered by the main testing criteria. Experiments on the *Siemens* suite manifest the applicability of the proposed approach and present interesting insights into the use of cluster analysis to the bi-criteria test suite reduction.
Keywords: Software regression testing, testing criteria, test suite minimization, test suite reduction, fault detection effectiveness

1 Introduction

Software regression testing is a critical activity in the development and maintenance of evolving software. A major difficulty with the regression testing is the excessive number of test cases accumulated while generating new test cases to test any new or modified functionality within the program [1]. To resolve the difficulty and to reduce the excessive cost of regression testing, various *coverage-based* and *distribution-based* techniques have been proposed [2,3,4,5,6,7,8]. These techniques attempt to permanently discard redundant test cases and retain the most effective ones to reduce the excessive cost of regression testing [6]. An effective technique for regression testing is to find a minimal subset of test cases which satisfy all the test requirements as the original set does [9]. A suitable subset could be found during the test case generation or after creating the test suite. Apparently the less the number of test cases the less time it takes to test the program. This consequently improves the effectiveness of the test process. The technique is commonly known as *test suite reduction* or *test suite minimization* in the literature and the resulting suite is called *representative set* [3].

Almost all the previous test suite reduction techniques could significantly reduce the size of the test suites. But an important issue deals with how well these reduced suites can be compared with their corresponding un-reduced suites using criteria rather than the suite size criterion. Since the purpose of test case execution is to detect faults

in the software, one measure of the suite quality is its fault detection capability. In fact, a potential drawback observed in test suite reduction studies is that permanent removal of test cases from a test suite may highly decrease the fault detection effectiveness of the remaining suite. Thus, the tradeoff between the time required to execute and manage test suites and their fault detection effectiveness should be considered when applying test suite reduction techniques [2].

Coverage-based minimization techniques attempt to eliminate redundant test cases while maximizing the code coverage of a program in order to ensure that all different execution paths are exercised and the majority of the faults are revealed. However, test cases considered as redundant in respect with the main testing criterion may be considered as essential test cases with respect to the additional criteria. This is evidenced by the results of the empirical studies presented in [10]. These results suggest that using several coverage criteria, rather than a single criterion, during test suite reduction is indeed useful in determining test cases which are likely to expose different faults. However, code coverage solely is not a sufficient criterion for selecting representative test cases from the original test suite. This is because high code coverage may be achieved by selecting simple test cases that do not reflect the program execution in real situations [11] and the resultant reduced suite may not be efficient in detecting faults.

A number of recent papers have investigated techniques for test suite reduction using *execution profiles*, applied in *distribution-based* techniques [8]. An execution profile generated by a test case identifies the program elements covered by the test case. For example, a profile may indicate which branches or data flow relationships are exercised by a test case. Distribution-based techniques reflect additional information about test cases that can be helpful in determining effective test cases when reducing the original test suite. Distribution-based test suite reduction techniques are examples of *observation-based* testing [12]. Observation-based testing is potentially applicable in situations where it is desirable to filter a large amount of test cases identifying a promising subset of test cases for conformance to the requirements [13]. Despite the effectiveness in detecting failures, observation based methods do not necessarily provide full coverage [8]. Moreover, empirical studies [8] show that the functionality of these methods varies with the size of the test suite. These studies also indicate that coverage-based and distribution-based techniques often do not select similar test cases and are complementary because they find different faults.

To address the limitations of the coverage-based and distribution-based approaches, in this paper a bi-criteria heuristic algorithm is proposed. The algorithm is applied to each cluster of execution profiles of the test cases, iteratively. In each of its iteration, a representative test case enhancing the cumulative coverage of the reduced test suite is selected from each cluster. The selection is based on the two criteria to improve the fault detection effectiveness. In fact, by selecting a test case from each cluster, all test cases that are now redundant with respect to the first criterion, are re-verified. Among such test cases, those that cover the maximum number of unmarked requirements with respect to the second criterion are inserted into the reduced suite. At the end of this process, the reduced suite is verified to be coverage adequate with respect to the second criterion. If not, the described process is repeated considering the second criterion, yielding bi-criteria coverage adequacy of the reduced suite. In order to evaluate the ap-

plicability of the proposed approach, we conducted experiments on the *Siemens* suite. We also implemented the well-known H algorithm [3], to compare the results of *reducing* suites using our distribution-based heuristic algorithm with those of *minimizing* test suites using the H algorithm.

The rest of the paper is organized as follows: Section 2 discusses the background of the test suite reduction techniques. Section 3 contains the outline of the proposed approach. Section 4 describes the empirical studies and the obtained results. Finally, conclusions are mentioned in section 5.

2 Background and Related Work

In this section, first the formal definition of the Test Suite Reduction problem will be introduced. Then, some related studies in this context are described. Afterwards, two general approaches to this problem are presented.

2.1 Test Suite Reduction

The first formal definition of test suite reduction problem introduced in 1993 by Harrold et al. [3] as follows:

Given. $\{t_1, t_2, \ldots, t_m\}$ is test suite T from m test cases and $\{r_1, r_2, \ldots, r_n\}$ is set of test requirements that must be satisfied in order to provide desirable coverage of the program entities and each subsets $\{T_1, T_2, \ldots, T_n\}$ from T are related to one of r_is such that each test case t_j belonging to T_i satisfies r_i.

Problem. Find minimal test suite T' from T which satisfies all r_is covered by original suite T.

Generally the problem of finding the minimal subset T', $T' \subseteq T$ which satisfies all requirements of T, is NP-complete [15], because we can reduce the *minimum setcover* problem to the problem of test suite minimization in polynomial time. Thus, researches use heuristic approaches to solve this problem. One heuristic method proposed by Harrold et al. [3], tries to find the smallest representative set that provides the same coverage as the entire test suite does.

Related work in the context of test suite reduction can be classified into two main categories: The works in which a new technique is presented [3,4,5,16,9,6,7], and empirical studies on the previous techniques [1,2,15,17]. The works which propose a new approach commonly include heuristic algorithms [14], genetic algorithm-based techniques [18] and approaches based on integer linear programming [19].

Testing criteria are defined in order to help the selection of subsets of the input domain to be covered during testing. For example, a code coverage criterion provides test suite adequacy with respect to coverage of the program entities and also provides a check on its quality. Assuming testing criterion C which satisfies by the test suite T, a test case, t, is *redundant* if the suite $T - \{t\}$ also satisfies C [5]. Therefore, removing those test cases which are redundant with respect to some specific criteria preserves test suite's adequacy with respect to it. In previous empirical studies [15] researchers commonly apply various code coverage criteria in their reduction techniques. The results of empirical studies [2,17] show that the more percentage of the test suite size is reduced the more percentage of faults will be lost.

2.2 Coverage-based Techniques

The main purpose of traditional software testing is to achieve the maximum code coverage [8] and the main idea is that the test suite is capable of exercising the whole program under test. In other words, there must be some test cases per program element (including statement, branch, def-use pair and so on) which can cover it. This is due to unreliability of the uncovered parts of the program during testing. But as mentioned above, code coverage alone is not sufficient for selecting test cases. If the code coverage is the main factor in determining the quality of a test suite, then it is desirable to have the minimal subset of the test suite that covers the program elements the same as the original suite [8]. Even if the code coverage is not the main factor in test suite quality, it is possible to use it as the first step of other reduction approaches. The main purpose of executing test cases is to expose faults as far as possible. Thus fault detection effectiveness of a test suite is of great importance. On the other hand, due to large amount of test cases and time constraints for retesting the software every time it modifies, it is necessary to keep the size of the test suite manageable [2]. Therefore, there must be some tradeoff between the suite size and its fault detection effectiveness.

Usually a test suite reduction technique attempts to remove redundancy among test cases and retain the most *effective* ones into the test suite [6]. Effective test cases are those that are capable of satisfying the most requirements as well as exposing the most of the existing faults. Note that the more requirements are satisfied, the more execution paths within the program would be exercised which yields the all kinds of the faults to be exposed. When a reduction algorithm determines some test cases to be redundant, it uses a certain criterion to do this. But redundant test cases which are thrown away during test suite reduction may exercise unique situations in the software with respect to another criterion [10]. For example, a reduction algorithm may select test cases which cover the most number of branches of a program. During the reduction, some test cases may cover the same branches as covered by test cases in the reduced suite. Hence, such test cases are redundant and will be discarded. However, such test cases may not be redundant with respect to a second criterion such as def-use pair. So they will be preserved from discarding because they cover execution paths which may contain some kind of faults. These results are evidenced by empirical studies in [10]. Therefore, using multiple criteria during reduction is useful to select test cases that are more likely to expose faults.

2.3 Distribution-based Techniques

Distribution-based techniques select test cases according to how their profiles are distributed by *dissimilarity metric* in the multi-dimension *profile space* [8]. Dissimilarity metric is a function that produces a real number for each pair of profiles which shows the degree of their dissimilarity. Distribution-based methods can be described using two closely related techniques: *cluster filtering* and *failure-pursuit*. Cluster filtering calls for using *automatic cluster analysis* to partition the test pool [8]. The purpose of cluster analysis is to partition the population such that objects with similar attributes are in the same cluster. After clustering, test cases are sampled from each cluster.

Distribution-based test suite reduction techniques are examples of *observation-based* testing [8]. It calls for first taking an existing set of test cases (possibly a large amount), and then executing them on an instrumented version of the program under test in order to obtain execution profiles that characterize the executions [13]. Next, it analyzes the profiles and selects a subset of test cases for conformance to requirements. The analysis of profiles can be achieved using *multivariate data analysis* like *clustering* [12]. Such kind of analysis can extract additional information from the profile data. In general, observation-based testing is applicable when it is desirable to identify a promising subset of test cases for conformance to requirements as well as filtering legacy test suites. An observation-based testing is defined by determining the following: (1) the type of the execution profile (2) a filtering or reduction technique to select a subset of executions and (3) an optional augmentation procedure. The type of the applied profiling should reflect the runtime events. Software testing research and practice suggest a variety of alternatives such as statement, basic block, branch, all-uses and so on [20].

The distribution-based techniques used for reducing test suites, determine some characteristics of the distribution profile that are likely to detect faults, and apply them during the selection of test cases. For example, clusters that have similar profiles contain redundant test cases and selecting one or a few representatives are sufficient. Also isolated profiles contain test cases that indicate unusual conditions and are likely to detect faults. Test case cluster filtering is defined by selecting *clustering algorithm, dissimilarity metric, number of clusters* and *sampling method* [8].

3 The Proposed Approach

Our approach to test suite reduction has been motivated by combining the two general techniques called distribution-based and coverage-based techniques. These techniques are fully described in the previous section. *Distribution-based* techniques are capable of determining similar test cases by means of clustering. These techniques cluster test cases within a test suite to determine test cases which exercise similar execution paths within the program. Since all the test cases within a same cluster have almost a same execution profile, most of them are likely to be redundant. Hence, sampling some representatives from each cluster may lead to a selection of distinguished test cases which test different execution paths within the program under test. Therefore, the major benefit of these techniques is to compose a reduced suite of distinct test cases with minimum redundancy. However, despite of covering rather non-overlapping execution paths, distribution based techniques do not necessarily provide full coverage of the execution paths. On the other hand, the coverage-based techniques insist on test suits with full coverage rather than non-overlapping test cases. Therefore, we combined these two techniques to provide a new bi-criteria technique to form full coverage reduced test suites with minimum overlap in the execution profiles of the including test cases.

The more the number of testing criteria the bigger the size of the test suite will be. However, experimental results show that two testing criteria can establish an appropriate tradeoff between the suite size and the fault detection capability of the reduced test suite. The general procedure of our approach is as follows: At the first step, the program is instrumented according to two different coverage criteria. Then, test cases in the test

suite are executed over the instrumented version of the program to collect execution profiles. Next, the collected profiles are clustered by means of a clustering algorithm. The clustered test cases and the testing requirements are applied as inputs to run our test suite reduction algorithm. After the reduction, the reduced suite will be coverage adequate with respect to the first criterion. The adequacy of the reduced suite is verified with respect to the second criterion. In the case of inadequacy, a similar algorithm using the second criterion can be applied to augment test cases providing coverage adequacy of the reduced suite with respect to the second criterion.

A detailed pseudocode description of the proposed algorithm is given in Fig. 1 and 2. Initially, all the testing requirements are unmarked and the number of unmarked requirements considering each of the two testing criteria is counted for each test case. Then, all the clusters of the test cases are sorted in ascending order of their number of test cases. In the next step, the algorithm keeps looking for representative test cases in each cluster of the sorted list of clusters until the selected representative test cases form a coverage adequate test suite with respect to the first criterion. In each of its iteration of the while-loop in Fig. 1, all the test cases within the next cluster of the test suite are added to a set called *list*. A function, *SelectTest*, selects a representative test case from the cluster and augments it to the reduced suite, *RS*. Each testing requirement satisfied by the selected representative test case is marked as satisfied in two arrays called $marked_1$ and $marked_2$. Each of these marked requirements may be satisfied by a number of test cases. If all the requirements satisfied by each of these test cases are marked with respect to one of the two test criteria, the test case number is augmented to the list of the redundant test cases, redundanti, for the ith criterion. Finally, all those cases which are in the current cluster, *list*, and their numbers appear in the set $\{redundant_1 - redundant_2\}$, are augmented to a set, *redundant*. The function *SelectSomeMoreTestCases* is then invoked to select some test cases from the set, *redundant*, and add the selected test cases to the final reduced suite, *RS*.

The function *SelectTest* selects a test case from a certain cluster. This function attempts to find a test case with maximum number of unmarked requirements with respect to the first criterion. In case of a tie, the number of unmarked requirements for the second criterion is considered. If tie occurs again, sum of the satisfied requirements with respect to the both criteria is considered, and finally a test case is selected randomly. The function *SelectSomeMoreTestCases* attempts to find some test cases from *redundant* that satisfy the maximum number of unmarked requirements with respect to the second criterion. Each time a test case is selected those test cases that become redundant are removed from the *redundant* set.

4 Empirical Studies

To evaluate and compare the proposed approach with prior approaches in test suite reduction area, an experiment similar to previous studies has been conducted. This section describes the experiment.

```
define:
    requirement^i: set of coverage requirements for minimization with respect to the ith criterion:
 r_1^i, r_2^i, ..., r_n^i, i = 1,2
input:
    t_1,t_2,...,t_m: test cases present in the unreduced test suite
    cv_i[m,n]: coverage matrix representing the ith requirement coverage of each test case
    clusters_i: array[1..k] of cluster instances for the ith requirement, each containing similar test cases
output:
    RS: a reduced suite of test cases from the original test suite.
algorithm TestSuiteReduction
begin
    currentClusterIndex := 0; redundant := redundant1 := redundant2 := { }; RS := { }; // initialization
    foreach criterion i and requirement j do marked_i[j] := false;
    for each test case t and criterion i do
      numUnmarked^i[t] := number of the elements in cv_i[t,n] for which cv_i[t,j] == true;
    Sort(clusters,ascending); // sorts the input array ascending or descending
    while there exists r_i^1 ∈ requirements^1 s.t. marked_1[i] == false do
      if currentClusterIndex == k then currentClusterIndex := 0;
      list := all t_j ∈ clusters_1[currentClusterIndex];
      if Card(list) == 1 then
        test := t ∈ list;
        if exists r_j^1 ∈ requirements^1 s.t. cv_1[test, r_j^1] == true and marked_1[j] == false then
          nextTest := test;
        else nextTest := 0;
        endif
      else nextTest := SelectTest(list);
      endif
      if nextTest ≠ 0 then
        RS := RS ∪ {nextTest};
        foreach criterion c do
          foreach r_j^c ∈ requirements^c s.t. cv_c[nextTest, r_j^c] == true and marked_c[j] == false do
            marked_c[j] := true;
            foreach test case t in the test suite s.t. cv_c[t, r_j^c] == true do
              numUnmarked^c[t] := numUnmarked^c[t] – 1;
              if numUnmarked^c[t] == 0 and t ∉ RS then redundant_c := redundant_c ∪ t;
            endfor
          endfor
        endfor
      currentClusterIndex := currentClusterIndex + 1;
      redundant := redundant_1 - redundant_2;
      if Card(redundant) != 0 then
        redundant := redundant ∩ list;
        SelectSomeMoreTestCases(redundant, RS);
        redundant := redundant_1 := redundant_2 := { };
      endif
    endwhile
    return RS;
end TestSuiteReduction
```

Fig. 1. Pseudocode description of our proposed heuristic algorithm for test suite reduction.

4.1 Subject Programs and Measures

We used the *Siemens suite* as subjects in our experiments. Siemens suite includes seven programs in C language developed by the researchers at Siemens Corporation for exper-

```
function SelectTest(testCaseSet)
begin
   foreach t_i in testCaseSet do compute numUnmarked^1[i], the number of elements in cv_1[t_i, r_j^1] for which
            marked_1[j] == false;
   testList1 := all t_i from testCaseSet for which numUnmarked^1[i] is the maximum;
   if Card(testList1) ≠ 0 then
      if Card(testList1) == 1 then testCase := the test case in testList1;
      else
         foreach t_i in testList1 do compute numUnmarked^2[i], the number of elements in cv_2[t_i, r_j^2] for which
                  marked_2[j] == false;
         testList2 := all t_i from testList1 for which numCovered^2[i] is the maximum;
         if Card(testList2) == 0 then testList2 := testList1;
         if Card(testList2) == 1 then testCase := the test case in testList2;
         else
            foreach t_i in testList2 do compute numCovered[i], sum of the the number of elements for which
                     cv_1[t_i, r_j^1] == true or cv_2[t_i, r_k^2] == true;
            testList3 := all t_i from testList2 for which numCovered[i] is the maximum;
            if Card(testList3) == 0 then testList3 := testList2;
            if Card(testList2) == 1 then testCase := the test case in testList3;
            else testCase := any test case in testList3;
            endif
         endif
      endif
   endif
   return testCase;
end SelectTest
```

```
function SelectSomeMoreTestCases(redundant, RS)
begin
   foreach t_i in redundant do compute moreCov[i], the number of elements in cv_2[t_i, r_j^2] for which
            marked_2[j] == false;
   while there exists t_i ∈ redundant s.t. moreCov[i] > 0 do
      test := any test case t_i from redundant for which moreCov[t_i] is the maximum;
      RS := RS ∪ {test}; redundant := redundant – {test}; redundant2 := { };
      foreach r_j^2 ∈ requirements^2 s.t. cv_2[test, r_j^2] == true and marked_2[j] == false do
         marked_2[j] := true;
         foreach test case t in the test suite s.t. cv_2[t, r_j^2] == true do
            numUnmarked^2[t] := numUnmarked^2[t] – 1;
            if numUnmarked^2[t] == 0 and t ∉ RS then redundant2 := redundant2 ∪ t;
         endfor
      endfor
      redundant := redundant – redundant2;
      foreach t_i in redundant do compute moreCov[i], the number of elements in cv_2[t_i, r_j^2] for which
               marked_2[j] == false;
   endwhile
end SelectSomeMoreTestCases
```

Fig. 2. Function *SelectTest* to select a test case from current cluster and function *SelectSomeMoreTestCases* to select some test cases from the *redundant* set into the *RS*.

iments with control-flow and data-flow test adequacy criteria [21]. These programs are associated with several faulty versions. Each faulty version of each program contains a single fault seeded in it. For each program there is a test pool which contains test cases developed for different black-box and white-box testing objectives.

To investigate the effectiveness of our approach, we measured the following from our experiments:

1. The *percentage suite size reduction* $= \frac{|T|-|T_{Red}|}{|T|} \times 100$, where $\mid T \mid$ is the number of test cases in the original test suite and $\mid T_{Red} \mid$ is the number of test cases in the reduced test suite.
2. The *percentage fault detection loss* $= \frac{|F|-|F_{Red}|}{|F|} \times 100$, where $\mid F \mid$ is the number of distinct faults exposed by original test suite and $\mid F_{Red} \mid$ is the number of distinct faults detected by the reduced suite.

4.2 Clustering and Analysis Tools

In order to cluster execution profiles of test cases, Weka 3.5.8 [22] has been used. Weka is a java-based tool that provides a uniform interface to many different learning algorithms along with methods for pre and post processing and for evaluating the results of learning schemas on any given dataset. It also provides implementations of the state-of-the-art learning algorithms and has been tested under Linux, Windows and Macintosh operating systems. Besides, we used SAS 9.1.3 [23] to create boxplots. Boxplot diagrams are commonly used to visualize the empirical results in test suite reduction studies.

Our experiments include a large number of test cases each of which covers a lot of requirements. Thus, a large number of high dimensional data should be clustered. However, fast and effective clustering of such data is extremely difficult because of their large volume and high dimensionality [24]. On the other hand, clustering algorithms need to set the number of clusters. Selecting the number of clusters affect the separation of test cases. The *CLOPE* algorithm [24] is a fast and efficient method for clustering large and high dimensional data. In this algorithm, a value called *Repulsion* is set which controls level of intra-cluster similarity. The number of clusters changes by varying this value. We felt that as a first step for exploration, using the *CLOPE* algorithm would lead to a good compromise between fast and effective clustering and a reasonable number of clusters.

4.3 Experiment Setup and Results

Our experiments follow a setup similar to that used by Rothermel et al [2]. For each program, we created branch coverage adequate test suites for six different suite ranges named as B, B1, B2, B3, B4 and B5. For each suite range, we first selected $X* LOC$ test cases randomly from the test pool and added to the test suite, where X is 0, 0.1, 0.2, 0.3, 0.4 and 0.5 respectively and *LOC* is the number of lines of code for each program. Then, randomly-selected test cases are added into the test suite as necessary so long as each test case increased the cumulative branch coverage of the suite, until the test suite becomes adequate with respect to branch coverage. In this way, the developed test suites have various types and varying levels of redundancy exist between them. We created 1000 such branch coverage adequate test suites for each program. In order to gather branch coverage information of test cases, all programs were hand-instrumented.

In our experiments, we used all-uses coverage as the second criterion. We measured all-uses coverage using the ATAC tool [25].

In order to evaluate the effectiveness of our approach, we implemented our algorithm and applied it to the generated suites with branch coverage as first criterion and all-uses as the second criterion. We also implemented the H algorithm [3] and conducted similar experiments on the generated suites with respect to branch coverage. The results of this experiment are shown in the columns labeled H and P in Table 1 for the H algorithm and the proposed algorithm respectively. The values in each row of the table are average values for 1000 suites in each range. In this table, $\mid T \mid$ indicates for the original suite size, $\mid F \mid$ for the number of faults exposed by the original suite, $\mid T_{Red} \mid$ for the reduced suite size, $\mid F_{Red} \mid$ for the number of faults exposed by the reduced suite size, %Size Reduction for the percentage suite size reduction and %Fault Loss for the percentage fault detection loss.

The boxplot in the Fig. 3 shows the distribution of the percentage of size reduction (SR) and percentage fault detection loss (FL) in the largest suite size range (B5) for each program. In this figure, boxes are paired such that, white pair of boxes shows percentage of size reduction and gray pair of boxes shows the percentage of fault detection loss. In each pair, left side box indicates for our algorithm and the right side one indicates for the H algorithm. For all programs, it is observed that the percentage suite size reduction by the proposed algorithm is less than the respective suite size reduction using the H algorithm. This result is expected since our approach selects some additional test cases that are not *redundant* with respect to the second criterion though they are redundant with respect the first criterion. For all programs, the percentage fault detection loss of the suites reduced by the proposed algorithm is less than the respective suites reduced by the H algorithm. We reason that the reduced suites generated using the proposed algorithm, are more efficient due to two main causes. First, clustering results in partitioning the test cases such that those with different execution profiles are gathered into different clusters. This gives us the opportunity to select as much unique test cases as possible.

Second, using second criterion during reduction may cause to selecting redundant test cases with different coverage of execution path within the program. These are also evidenced by the results of empirical studies.

To determine whether the degradation in fault detection loss observed for the proposed algorithm over the H algorithm is statistically significant, we conducted a *hypothesis test for the difference of the two means* [26]. The samples are the number of distinct faults exposed by each of the 1000 reduced test suites for suite size range B5 using the H algorithm and the proposed algorithm. We considered the *null hypothesis* that there is no difference in the mean number of the exposed faults by the two algorithms. We used a reference table of critical values presented in [26]. Table 2 shows the resulting z *values* computed for the hypothesis test along with the percentage confidence with which we may reject the null hypothesis. Note that for z values greater than 4.0, we can reject the null hypothesis with the confidence over than 99.99 percent.

4.4 Threats to Validity

In this section, we describe the potential threats to validity of our study.

Table 1. The results of the experiments comparing the H algorithm and the proposed algorithm.

Program	Suite size range	$\mid T \mid$	$\mid F \mid$	$\mid T_{Red} \mid$		$\mid F_{Red} \mid$		% Size Reduction		% Fault Loss	
				H	P	H	P	H	P	H	P
printtokens	B	17.79	3.85	8.19	8.48	3.26	3.56	52.86	51.13	9.21	6.39
	B1	29.4	4.01	8.22	8.64	3.23	3.51	69.13	67.66	14.66	10.80
	B2	46.45	4.29	8.15	8.73	3.16	3.56	77.48	76.17	19.31	15.04
	B3	65.48	4.53	8.05	8.72	3.25	3.57	82.58	81.41	23.42	19.08
	B4	84.96	4.68	7.97	8.80	3.03	3.53	85.63	84.51	28.99	22.00
	B5	107.63	4.92	7.98	8.95	3.07	3.61	87.90	86.89	33.97	24.06
printtokens2	B	13.41	8.47	8.01	8.99	8.17	8.27	39.08	31.62	4.46	2.28
	B1	28.79	8.66	7.63	9.19	8.04	8.37	67.11	61.16	9.96	5.44
	B2	51.68	9.24	7.13	9.40	8.10	8.57	79.47	74.57	12.88	6.99
	B3	75.1	9.39	6.90	9.82	8.12	8.64	84.06	79.65	15.23	7.87
	B4	99.73	9.53	6.64	10.07	8.19	8.75	87.74	83.69	16.78	8.04
	B5	128.01	9.68	6.41	10.32	7.90	8.76	90.76	87.10	18.09	9.36
replace	B	22.97	13.53	13.65	15.71	10.85	11.83	40.02	31.13	19.17	11.95
	B1	38.36	15.49	13.52	16.11	11.01	12.23	61.15	53.99	29.92	20.22
	B2	59.55	17.11	13.40	16.54	10.90	12.44	71.33	65.39	44.79	25.89
	B3	84.94	18.64	13.18	16.84	9.97	12.83	78.28	73.24	49.69	29.94
	B4	108.98	19.45	12.99	17.02	09.75	12.83	81.67	77.13	52.96	32.51
	B5	131.09	20.34	12.82	17.04	08.65	13.11	84.22	80.18	55.99	34.00
schedule	B	8.73	3.54	6.30	6.8	2.97	3.20	26.65	21.09	14.22	8.39
	B1	18.82	4.57	6.19	7.16	2.89	3.30	60.46	55.05	34.36	25.82
	B2	33.19	5.33	6.01	7.23	2.96	3.39	74.66	70.46	43.03	35.04
	B3	46.76	5.74	5.96	7.32	3.02	3.49	79.81	76.22	46.02	37.87
	B4	61.67	5.97	5.86	7.33	2.92	3.43	83.30	80.12	50.04	41.25
	B5	76.22	6.10	5.84	7.28	2.85	3.38	85.56	82.49	51.91	43.30
schedule2	B	7.96	2.39	5.35	5.66	2.09	2.15	31.18	27.37	12.65	9.53
	B1	18.06	2.77	5.24	5.77	2.08	2.17	64.17	60.86	21.99	17.97
	B2	32.08	3.44	5.09	5.73	2.10	2.27	77.03	74.46	35.49	29.68
	B3	44.69	3.88	5.00	5.77	2.03	2.39	80.81	78.63	39.96	33.03
	B4	59.86	4.36	4.88	5.68	2.12	2.40	85.24	83.46	45.78	39.71
	B5	74.77	4.86	4.72	5.61	2.06	2.51	87.77	86.04	51.94	43.38
tcas	B	5.70	7.13	5.00	5.04	6.44	6.59	11.12	10.45	8.32	6.57
	B1	9.22	8.70	5.00	5.10	6.09	6.52	39.53	38.56	26.04	21.48
	B2	15.16	11.21	5.00	5.09	6.37	6.53	57.70	56.97	42.41	35.96
	B3	21.27	13.61	5.00	5.08	6.58	6.68	66.06	65.50	52.95	43.98
	B4	28.29	15.54	5.00	5.06	6.57	6.77	71.99	71.60	61.19	50.08
	B5	35.70	17.28	5.00	5.05	4.74	6.64	76.35	76.05	64.06	55.13
totinfo	B	9.02	13.85	5.54	6.34	12.89	13.17	36.97	27.28	6.69	3.84
	B1	20.62	14.78	5.46	6.49	11.83	13.62	68.07	60.52	17.71	10.59
	B2	36.74	16.11	5.33	6.51	11.91	13.86	78.66	73.68	25.99	17.06
	B3	52.50	17.04	5.25	6.37	11.17	13.97	83.43	80.64	31.38	21.40
	B4	70.25	17.77	5.20	6.33	10.86	14.10	87.10	83.66	32.09	22.87
	B5	86.23	19.37	5.18	6.28	13.00	14.01	88.23	86.22	31.79	22.45

Table 2. Computed z value and the corresponding percentage of confidence for rejecting the null hypothesis for each program.

Program Name	Computed z value	Percentage of confidence for rejecting the null hypothesis
printtokens	15.46	>99.99%
printtokens2	15.36	>99.99%
replace	39.45	>99.99%
schedule	8.82	>99.99%
schedule2	8.20	>99.99%
tcas	16.01	>99.99%
totinfo	7.66	>99.99%

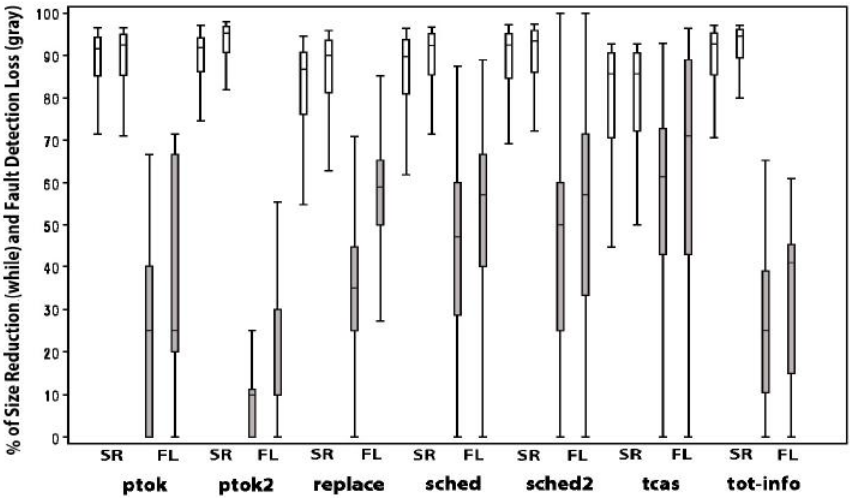

Fig. 3. The boxplot for the percentage suite size reduction and percentage fault detection loss.

Threats to construct validity. In our study, the measurement for the percent of fault loss assumes simple model for cost which treats all faults as equally severe. But in practice, faults have wide range of severity from less critical to more critical.

Threats to internal validity. The most important issue deals with hand-instrumentation of code which we have done for obtaining branch coverage of test cases. To validate the correctness of this process, the instrumentation was re-verified. Another issue is composition of the test suites. However, we utilized the process of creating suites which was employed in previous studies [2].

Threats to external validity. Siemens programs are widely used in software testing studies. However, these programs are limited and their faults are known. Moreover, they are not real programs and the faults are hand-seeded. Space program is a big and real one, used to our case study. But it is one of such a program we used.

5 Conclusions

We have presented a new approach for test suite reduction that attempts to identify effective test cases in terms of their fault detection effectiveness. This approach is based on the intuition that *clustering* of the test cases *execution profiles*, can partition the test suite such that similar test cases in terms of exercising a certain coverage criterion, would be in the same clusters. Hence, determining redundant test cases from essential ones could be done effectively and simply. In order to improve the fault detection effectiveness of the reduced suite, we used two coverage criteria during the reduction process. The results of empirical studies demonstrate the effectiveness of the proposed approach by generating reduced test suites with less fault detection loss.

References

1. Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical Studies of Test-Suite Reduction. Journal of Software Testing, Verification, and Reliability 12(4), 219–249 (2002)
2. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society (1998)
3. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for Controlling the Size of a Test Suite. ACM Transactions on Software Engineering and Methodology 2, 270–285 (1993)
4. Chen, T.Y., Lau, M.F.: Heuristics Toward the Optimization of the Size of a Test Suite. In: Proc. of the 3rd International Conference on Software Quality Management, vol. 2, pp. 415–424. Seville, Spain (1995)
5. Jones, J.A., Harrold, M.J.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering 29, 195–209 (2003)
6. McMaster, S., Memon, A.: Call-Stack Coverage for GUI Test Suite Reduction. IEEE Transactions on Software Engineering 34, 99–115 (2008)
7. Tallam, S., Gupta, N.: A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In: Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, Lisbon, Portugal (2005)
8. Leon, D., Podgurski, A.: A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In: Proc. of the 14th International Symposium on Software Reliability Engineering. IEEE Computer Society (2003)
9. Chen, Z., Xu, B., Zhang, X., Nie, C.: A Novel Approach for Test Suite Reduction Based on Requirement Relation Contraction. In: Proc. of the 2008 ACM Symposium on Applied computing. ACM, Fortaleza, Ceara, Brazil (2008)
10. Jeffrey, D., Gupta, N.: Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. IEEE Transactions on Software Engineering 33, 108–123 (2007)
11. Marick, B.: The Craft of Software Testing: Subsystem Testing. Prentice Hall, Englewood Cliffs, NJ (1995)
12. Leon, D., Podgurski, A., White, L.J.: Multivariate Visualization in Observation-based Testing. In: Proc. of the 22nd International Conference on Software engineering. ACM, Limerick, Ireland (2000)
13. Dickinson, W., Leon, D., Podgurski, A.: Finding Failures by Cluster Analysis of Execution Profiles. In: Proc. of the 23rd International Conference on Software Engineering. IEEE Computer Society, Toronto, Ontario, Canada (2001)

14. Dickinson, W., Leon, D., Podgurski, A.: Pursuing Failure: the Distribution of Program Failures in a Profile Space. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Vienna, Austria (2001)
15. Zhong, H., Zhang, L., Mei, H.: An Experimental Study of Four Typical Test Suite Reduction Techniques. Information and Software Technology 50, 534–546 (2008)
16. Chen, T.Y., Lau, M.: A New Heuristic for Test Suite Reduction. Information and Software Technology 40 (5-6) (1998)
17. Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of Test Set Minimization on Fault Detection Effectiveness. Software-Practice and Experience 28, 347-369 (1998)
18. Mansour, N., El-Fakih, K.: Simulated Annealing and Genetic Algorithms for Optimal Regression Testing. Journal of Software Maintenance 11, 19-34 (1999)
19. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-Criteria Models for All-Uses Test Suite Reduction.In: Proc. of the 26th International Conference on Software Engineering. IEEE Computer Society (2004)
20. Harrold, M.J., Rothermel, G., Wu, R., Yi, L.: An Empirical Investigation of Program Spectra. In: Proc. of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, Montreal, Quebec, Canada (1998)
21. Rothermel, G., Elbaum, S., Kinneer, A., Do, H.: Software-artifact Infrastructure Repository, http://www.cse.unl.edu/ galileo/sir
22. Witten, H. Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd ed., Morgan Kaufmann series in Data Management Systems (2005)
23. SAS 9.1.3 Documentation, SAS/GRAPH 9.1 Reference, http://support.sas.com/documentation/onlinedoc/91pdf/index_913.html
24. Yang, Y., Guan, X., You, J.: CLOPE: a Fast and Effective Clustering Algorithm for Transactional Data. In: Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, Edmonton, Alberta, Canada (2002)
25. Horgan, J R., London, S.A.: ATAC: A Data Flow Coverage Testing Tool for C. In: Proc. Symp. Assessment of Quality Software Development Tools, pp. 2-10 (1992)
26. Freund, J.E.: Mathematical Statistics. 5th ed. Prentice-Hall (1992)