TEST CLUSTER SELECTION USING COVER COEFFICIENTS

Mahadevan Subramaniam* and Parvathi Chundi*

Abstract

Clustering test profiles to retrieve relevant tests is a recurring theme in software validation. A novel clustering approach using a probabilistic notion of coverage among line-based test profiles is described. The approach automatically determines the number of clusters to generate a clustering and can potentially group together tests to execute a few distinct lines of code. A simple method to identify tests affected by program changes is developed and used to determine the retrieval effort needed for cluster-based retrieval of affected tests. The approach is applied to four unix utility programs from a popular testing benchmark. Our results show that comparing original and new clusterings with respect to test cases is a promising criterion for deciding the potential re-clustering points in software evolution.

Key Words

Software testing, data clustering, regression, observation testing

1. Introduction

Recently, there has been a lot of interest in using clustering algorithms to retrieve relevant tests from large test suites. It has been empirically observed that tests revealing the same fault usually exhibit similar runtime behaviours [1]. Clustering algorithms aim to leverage this correspondence by grouping tests having similar execution profiles. The generated test clusterings are sampled in different ways to retrieve relevant tests to improve several software validation activities including observation-based testing [2]–[4], regression test selection [5], [6], prioritization [7], minimization [8] as well as fault localization [9].

This paper considers the use of clustering algorithms for software regression testing where a new version of the program has to be re-tested to ensure that it is functioning correctly. In this case, the outcome of a subset of available tests, called *affected tests*, on the new version, is analysed for validation. Informally, affected tests are tests that execute a modified line of a program and/or whose outcomes are affected by a program change. Analysing the outcome of affected tests for conformance is usually more expensive than running the tests. Cluster-based retrieval uses clustering algorithms to group the affected tests based on their profiles so that the tests whose outcomes are to be analysed are judiciously retrieved.

In this context, after identifying the affected tests regression testing can be performed in two ways. Either the existing clustering that was generated using the original program version can be reused or a new clustering based on the test profiles obtained from the newer version can be generated. In both the cases, the affected tests in the clusterings are marked, and all the tests in every test cluster whose seed is an affected test are retrieved to analyse their outcomes.

A question that arises in the cluster-based retrieval of affected tests is – whether the affected test retrieval using the newer clustering structure is always better compared to that using the existing structure. It is likely that the existing clustering structure is not well suited to validate a new version as the test profiles over the new version are likely to be different than the existing ones, and this may cause the tests to be grouped differently than before. However, while the newer test clustering can be readily generated, it may not reflect the best grouping of affected tests in many situations. For instance, affected tests may get widely distributed over the new clustering or seed of a large cluster of highly similar unaffected tests may become affected. In such cases and others, retrieving affected tests using the existing clustering can lead to higher retrieval quality with a lower effort.

In this paper, we develop a cluster-based approach called the cover-coefficient clustering (C3) that retrieves tests having similar behaviours to the affected tests for software regression testing. It is inspired by the cover-coefficient concept originally developed by Can and Ozkarahan [10] to perform cluster-based retrieval of documents from text databases. A unique feature of this approach is that it automatically determines the number of clusters and the cluster seeds by analysing the input data. Similarity among tests is characterized in the C3 approach using an asymmetric, probabilistic notion of test coverage. We develop a simple procedure to identify affected tests by analysing test profiles and describe how the affected tests are used to decide on the best C3 for validating a new version.

The rest of the paper is organized as follows. In Section 2, a brief overview of program execution profiles,

* University of Nebraska Omaha, Omaha, NE 68182, USA; e-mail: {msubramaniam, pchundi}@unomaha.edu

Recommended by Dr. Shihong Huang
(DOI: 10.2316/Journal.213.2015.4.213-1071)

and clustering of data is given. Section 3 describes the C3 approach to partition a test suite using test execution profiles. Section 4 extends the approach to multiple versions. Procedures to identify affected tests by analysing execution profiles are presented. Experiments and results are presented in Section 5. Section 6 discusses related work. Section 7 concludes the paper.

2. Background and Preliminaries

2.1 Test Execution Profiles

The execution profile of a test represents the runtime behaviour of a test in the form of runtime events such as the lines, blocks of code, branches, functions, and paths [11]. This paper uses line-based test execution profiles collected using the *gcov* (*Gnu* coverage) tool. A line-based test profile is a collection of triples. The first element of the triple is the execution status, the second element is the source line number, and the last element is the code content. The execution status has value 0 for the lines executed 0 times by the test and has value 1 for the lines executed one or more times by the test. The execution status values from each test profile are collated into a bit vector that is indexed by the line numbers. The bit vectors from all test profiles are combined to form a Boolean matrix (called E matrix below) which represents the profile information for the entire test suite. The dimensions of the Boolean matrix are given by the number of distinct line numbers in all of the test profiles.

2.2 Clustering Methods

Clustering methods are widely used to identify similar objects in a given group of objects [12]. The given set of objects is typically partitioned into clusters, where each cluster consists of similar objects. There are two classes of clustering algorithms – iterative and non-iterative. An iterative clustering algorithm such as *K-means* creates an initial partitioning of objects into k clusters and iteratively improves these clusters until some error criterion is met. Iterative algorithms can be costly for large data sets and need users to predict the number of clusters, which may be difficult.

Non-iterative algorithms such as hierarchical agglomerative clustering and C3 compute a numerical value for how similar (or dissimilar) each pair of objects are, in the given group of objects. Hierarchical agglomerative clustering then starts with n groups (or clusters) each containing a single object, merges the groups with the highest similarity value, re-computes the similarity values with the merged group as one of the objects and repeats the process. Once objects are merged into groups, the step cannot be undone, hence the algorithm is non-iterative. The hierarchical agglomerative clustering process terminates when sufficient number of clusters are generated or objects are too far apart to be merged. The terminating condition is again hard for users to predict.

The C3 algorithm [10] used in this paper is non-hierarchical, non-iterative, and predicts the number of clusters from the characteristics of the given data set and performs very well in practice. The details of this algorithm are described in the next section.

3. Cover-Coefficient Clustering (C3)

The C3 algorithm that was originally proposed by Can and Ozkarahan [10] is used for clustering text databases using keywords. In this paper, we use the C3 algorithm to partition a group of tests into disjoint clusters based on their execution profiles. Consider a test suite T consisting of m tests whose execution profiles are bit vectors of size n. The input data for C3 is represented by an $m \times n$ Boolean matrix E consisting of these bit vectors. The rows of E denote the tests of T, and the columns of E denote the lines (of code) that are executed by at least one test of T. Without any loss of generality, we assume that each test in T executes at least one line that is, matrix E does not have any rows or columns consisting entirely of zero-valued entries.

A probabilistic notion of *cover coefficients* is used to identify relations among tests based on their execution profiles. Informally, the cover coefficient of a test with respect to another denotes the extent to which the execution profile of the first test is covered by that of the second one. Cover coefficient c_{ij} of a test t_i with respect to a test t_j is the probability that a line l_k executed by t_i is also executed by t_j. Let α_i and β_j are the reciprocals of the sum of the entries in the ith row and the jth column of the E matrix, respectively. The cover coefficient of row i with respect to row j is: $c_{ij} = \alpha_i \times r_{ij}, r_{ij} = \sum_{k=1}^{n} (E_{ik} \times \beta_k \times E_{jk})$.

The cover coefficients of the tests in a test suite T of size m is represented by a $m \times m$ matrix, C, whose elements are computed using the above equation. Note that the computation of the entry c_{ij} uses the profiles of all the tests. In particular, the value of c_{ij} does not equal the ratio of common number of lines executed by the two tests over the total number of lines executed by the test t_i.

The diagonal entry c_{ii} of the ith row is called the *decoupling coefficient* of that row. The decoupling coefficient of the C matrix, δ, is the mean value of the decoupling coefficients of its rows. If a test has a distinguishing profile, it executes lines distinct from other tests and hence its profile is not likely to be covered that of the other tests. Therefore, it may be necessary to place this test in a separate cluster. In general, we can estimate that the number of clusters should be high (low) when there are a large (small) number of distinguishing execution profiles, which is specified by the de-coupling coefficient δ. Let n_c be the number of test clusters of a test suite T. It is estimated to be $n_c = m \times \delta$.

Test suite T is partitioned into n_c (actually, $n_c + 1$ clusters as described below) clusters by first identifying *seed* tests from T that are sufficiently dissimilar and cover an adequate number of remaining tests. A single seed test is identified for each of the n_c clusters based on the *clustering power* of the tests. The clustering power of test t_i is: $P_i = c_{ii} \times (1 - c_{ii}) \times \sum_{k=1}^{n} E_{ik}$.

To partition the tests in T into the clusters, the cluster seeds are identified by ranking the tests in T based on their clustering power. The top n_c tests are chosen as cluster seeds and are assigned to one cluster each. Ties are broken arbitrarily. A test t_j is considered a false seed and eliminated if there exists a seed t_i such that the coefficients c_{ii}, c_{jj}, c_{ji} and c_{ij} are sufficiently close, that is, the magnitude of the pairwise difference of c_{ii}, c_{jj}, c_{ij} and c_{ji} are all within a specified threshold ϵ. In this case, the false seed is eliminated, and the next seed in the sorted order is picked. A threshold value of 0.001 was used based on the spread of the seed values. However, no false seeds were found in our experiments for our data set.

To populate the clusters, each remaining test t_i of T is assigned to a cluster whose seed t_j maximally covers t_i, that is, c_{ij} is a maximal value, for $1 \leq j \leq n_c$. If more than one seed maximally cover t_i, the test is assigned to the cluster whose maximal covering seed has the higher clustering power (ties are broken arbitrarily). If there exist tests in T that cannot be assigned to any of the clusters because none of the seeds cover them, that is, $c_{ij} = 0$ for all seed tests t_j, these tests are collected into a *ragbag* cluster, [$(nc+1)$th cluster].

4. Affected Tests

An existing clustering structure may not be well suited to validate a new program version as the test profiles over the new version are likely to be different than the existing ones, and this may cause the tests to be grouped differently than before. One can always generate a new clustering structure with each new program version. However, it is not necessary that the new structure should be better than the existing structure for validating the new version. In many cases, certain tests called *affected tests*, which are crucial for validating the new version, may be retrieved more easily using the existing structure in comparison to the new one. In such cases, performing cluster-based retrieval of tests using the existing structure may produce better results. Below, we describe a simple procedure to identify affected tests by analysing test profiles.

A test is affected if it executes a significant number of modified lines with respect to the original and new program versions and/or generates different outputs over these two versions. As tests executing large number of modified lines may produce the same output whereas different outputs may be produced while executing only a few modified lines, we use two values to identify affected tests – *modification-revealing* value (δ_r) and *modification-traversing* value (δ_t).

Let $R \subseteq T$ be the set of all modification-revealing tests in a test suite T. The modification-revealing value of an arbitrary set of tests A, $A \subseteq T$, $\delta_r(A) = \frac{|MR(A)|}{|R|}$, where $MR(A)$ is the number of modification-revealing tests of T appearing in A. The modification-traversing value of a test t for a program version v is $\delta_t(t,v) = \frac{\sum_{m \in M(v)} f_m}{\sum_{l \in L(v)} f_l}$, where line m is executed with frequency f_m and line l is executed with frequency f_l in the version v by the test t. The set $M(v)$ is the set of the modified lines of v, and $L(v)$ is the set of all lines of v. The numerator is the total number of times modified lines that are executed, and the denominator is the total number of times all the lines that are executed by the test t when it is run on v. The modification-traversing value of a test t in general, $\delta_t(t) = \max(\delta_t(t,o),\ \delta_t(t,n))$, is the maximum of the values over the original and new versions o and n. The modification-traversing value of a set of tests A, $\delta_t(A)$, is the average of the modification-traversing values of the tests in A.

Definition (Affected-tests). *Let $0 < \alpha,\ \beta \leq 1$, respectively, be the user specified modification-traversing and modification-revealing thresholds. The set of affected tests A is a subset of test suite T such that $\delta_t(A) \geq \alpha$ and $\delta_r(A) \geq \beta$.*

We choose the affected test set A that satisfies minimum requirements for both modification-traversing and modification-revealing values so that both modification-traversing/revealing tests are sufficiently represented in set A. As modification-traversing tests need not be modification-revealing, selecting tests solely based on the threshold α may produce an A not having modification-revealing tests. On the other hand, modification-revealing tests may have very low α value, and selecting tests solely based on threshold β may produce an A not having tests that execute a large number of modified lines. Neither outcome is helpful as it may cause the exclusion of certain desirable tests from A. Note that the set A may not exist for certain threshold values. Further, more than one choice of set A are feasible for a given pair of threshold values. Most modification-revealing tests are included in our experiments when the threshold β is close to the value 1 and α is a low positive value.

The main steps to find a set of affected tests of A from suite T for given α and β values are:

1. Perform a *diff* of the source files of the original and new versions and identify the modified lines.
2. Run test suite T on both versions to generate outputs and *gcov* files for each test for each version.
3. For each test t, compute $\delta r(t)$ using test outputs and $\delta t(t,\ v)$ using *gcov* files and modified lines.
4. Initially, let $A \subseteq T$ be a set of tests whose elements have a non-zero δt value. If $\delta r(A) < \beta$ then the set of affected tests is the empty set. Otherwise, sort A based on δt values.
5. Eliminate low δt valued tests that are not modification-revealing tests from the sorted list while maintaining $\delta t(A) \geq \alpha$. Then, eliminate low δt valued tests that are modification-revealing while maintaining $\delta t(A) \geq \alpha$ and $\delta r(A) \geq \beta$. Return the resulting set of tests A.

5. Experiments

The objective of our experiments was to study whether re-clustering must be performed for each new program version or the existing clustering can be re-used for economical retrieval of affected tests. Towards this goal, the proposed approach was implemented on a 12G RAM, quad-core machine running Ubuntu. We studied four programs from the software infrastructure repository (SIR) [13] – *Grep*,

Table 1
Unix Utility Programs from Benchmark

Program	No. of Lines	Non-faulty Ver.	Faulty Ver.	No. Faults	No. of Tests
Grep	10,929	$v1$–$v5$	$v6$–$v10$	20	470
Gzip	6,357	$v1$–$v5$	$v6$–$v10$	16	213
Sed	8,059	$v1$–$v7$	$v8$–$v14$	3	404
Space	9,126	$v2$–$v4$		0	500

Gzip, *Sed* and *Space* whose details are depicted in Table 1. The size of these programs ranged from 6,300 to 11,000 lines of C code. For all programs, version $v0$ was used as the base version. This version was modified by making code changes and/or injecting faults (pre-defined in file, FaultSeeds.h) to create several non-base versions. Five non-faulty and five faulty versions were created and analysed for *Grep* and *Gzip*, and seven non-faulty and seven faulty versions were created and analysed for *Sed*. The faulty versions were created by injecting, twenty faults in *Grep*, sixteen faults in *Gzip* and three faults in *Sed*. We analysed 38 non-faulty versions for *Space*. No faulty versions for this program were available in the benchmark. The program *Grep* was accompanied by two test suites in the benchmark. The larger test suite containing 470 tests was used. The program *Gzip* was accompanied by five test suites in the benchmark. The largest test suite comprised 213 tests was used. For the program *Sed*, the two available suites were joined to obtain a suite of 404 tests. A test suite with 500 tests was used for the program *Space*. The tests that produced different outputs compared to the base version were viewed as fault-revealing tests in every non-base version as no specifications were available in the benchmark.

5.1 Retrieval Effort

A random clustering, $R=\{\rho_1,\ldots,\rho_q\}$, corresponding to a C3, $G=\{G_1,\ldots,G_q\}$, has equal number and equal-sized clusters as the latter except for the tests being randomly assigned to the clusters in the former. Let $A=\{t_1,\ldots,t_k\}$ be the set of affected tests belonging to a test suite T. A test in T belongs to the test set A if and only if the test either has a profile containing a modified line or produces a different output compared to the base program version. Let $\pi_G(A)$ be the number of clusters in G that contains a test from A. Let $\pi_R(A)=P_1+P_2+\cdots+P_q$, where P_j denotes the probability that the cluster ρ_j will contain a test from A. The probability P_j is [10], [14],

$$P_j = \begin{cases} (1-\prod_{i=1}^{n} \dfrac{(m_j-i+1)}{(m-i+1)}, & k \le m_j \\ 1, & k > m_j \end{cases}$$

Above, $m_j = m$ – size of (ρ_j). If $k \le m_j$, the product computes the probability of choosing all the tests of the set A from clusters other than the cluster ρ_j. The complement of this product then gives the desired probability. If $k > m_j$, it is not possible to choose all the tests of A without the cluster ρ_j and hence $P_j = 1$. The *retrieval* of the test set A using G is more economical compared to R, if $\pi_G(A) \le \pi_R(A)$. Below, we will use $\pi(A)$ instead of $\pi_G(A)$ to refer to the retrieval effort using the C3.

To determine re-clustering points, our experiments compared the retrieval of a given affected test set A using the original (o) and the newer (n) C3s generated from the base and the non-base versions, respectively. Measures consisting of the values $\pi_o(A)$ and $\pi_n(A)$ across the two clusterings and $\delta_o = \pi_R(A) - \pi_G(A)$ for the base version and $\delta_n = \pi_R(A) - \pi_G(A)$ for the non-base version were used to compare the retrieval effort across versions. The retrieval effort using the original clustering is more economical if $[\pi_o(A),\ \delta_o(A)]$ is lexicographically lesser than or equal to $[\pi_n(A),\ \delta_n(A)]$, and *vice versa.*

5.2 Experimental Procedure

For each non-base version of each program, the following steps were carried out – (1) C3 and the corresponding random clustering were generated for both base and non-base versions. (2) Procedure to identify the set of affected tests, A, was invoked with the thresholds $\alpha=0.25$ and $\beta=0.75$. The set A consisted of tests that exhibited at least 25% difference over the test profiles when compared to the base version and included at least 75% of fault-revealing tests. (3) The differences over a test's profile were determined by comparing the corresponding rows in the E matrices of the non-base and base versions. (4) The economy of retrieval of affected tests was determined by computing the retrieval measures π and δ for both the base and the non-base versions and performing a lexicographic comparison.

5.3 Results

The results obtained for all the four programs are depicted in the four bar graphs in Fig. 1. The X-axes in these graphs specify the program version numbers, and the Y-axes specify the number of clusters retrieved from the clustering of the base version Go, the non-base version Gn, δs, for the base version do, and for the non-base version dn. Our results showed that retrieval using the base version is equally likely to be more economical than that produced using the newer versions; the retrieval using the newer version was more economical than that using the base version for the programs *Grep* and *Space* while that using the base version was more economical for the programs *Gzip* and *Sed*. We elaborate our results below.

For *Grep*, around 7–9 clusters were retrieved using original clustering, and around 5–9 clusters were retrieved using the newer clustering. Based on the π and δ values, it was concluded that retrieval using the original clustering had comparable performance with that using the newer clustering for $v3$ and $v5$, whereas for the remaining eight versions, using the newer clustering was more economical. The savings in the retrieval effort due to the

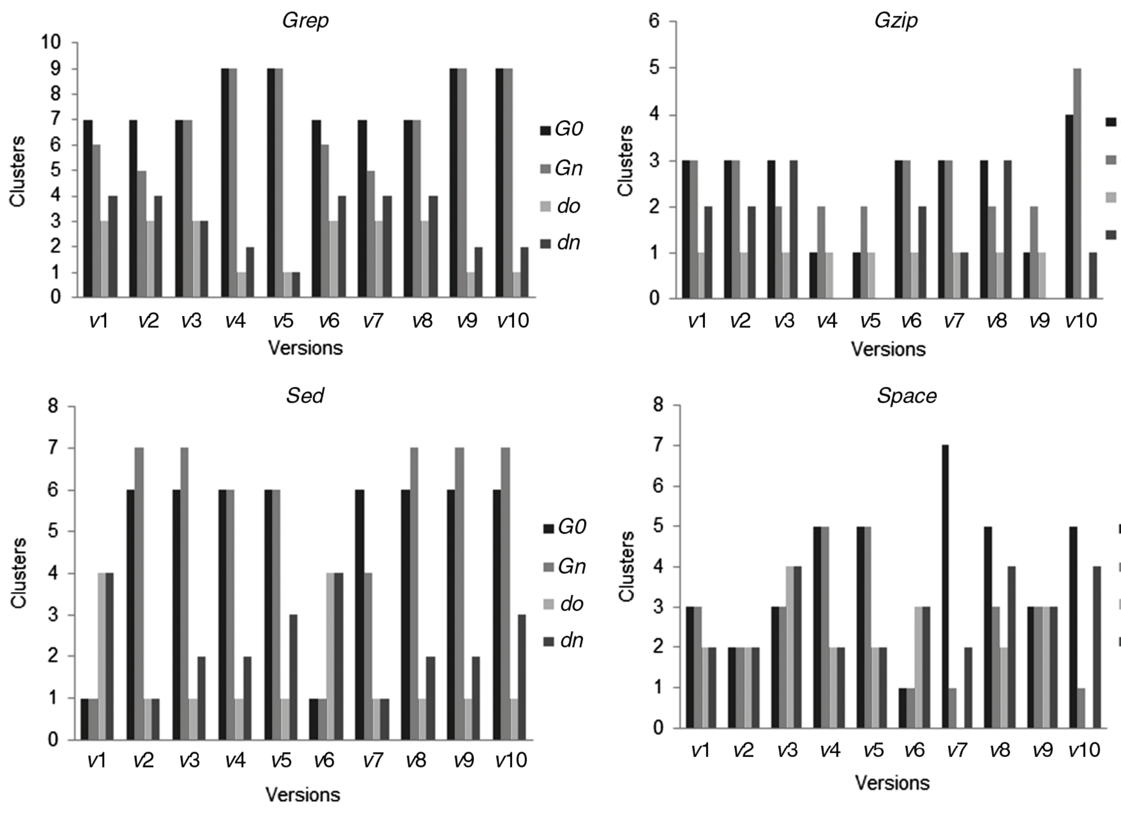

Figure 1. Test retrieval across base and non-base versions for Grep, Gzip, Sed and Space.

newer clustering ranged from 15% to 30% in these eight versions.

For *Gzip*, around 1–4 clusters were retrieved using the original clustering, and around 2–5 clusters were retrieved using the newer clustering. Retrieval using the newer clustering was more economical, for versions $v3$ and $v8$ with both the π and the δ values were better than their original clusterings. For the remaining eight versions, retrieval using the original clustering was more economical. The π values of the original clustering were the same or better in all these cases. The δ values of the original clustering were better in all cases except for version $v10$. Therefore, in all these cases, retrieval can be performed using the original clustering whereas re-clustering can be used for maintenance.

For *Sed*, about 1–6 clusters were retrieved using the original clustering, and around 1–7 clusters were retrieved using the newer clustering. The original and the newer clusterings have comparable retrieval performances for versions $v1$ and $v6$. The π and δ values were the same as those using the original clustering in these cases. Retrieval using the original clustering was more economical in all other cases. For versions $v4$ and $v5$, the π values were the same as those for the original clustering whereas the δ values for the original version were better. For the remaining six versions, the π values using the original clustering were better. These results suggest that for *Sed*, it may be better to retrieve tests using the original clustering and perform re-clustering primarily for maintenance in all these versions.

For *Space*, around 1–7 clusters were retrieved using the original clustering, and around 1–5 clusters were retrieved using the newer clustering. The retrieval effort using the newer clustering was better compared to that using the original clustering in all the 10 cases. For versions $v1$–$v6$ and version $v9$, the π values were the same as those for the original clustering. For versions $v7$, $v8$ and $v10$, the π value was significantly better compared to the original clustering. These results suggest that retrieval using the newer clustering leads to better performance in all the versions of *Space*.

5.4 Limitations and Threats to Experimental Validity

The threats to external validity are with respect to the generalizations of the results. We have not considered real industrial programs where the code base, platforms and versioning can be much more involved including branching. However, we have used the programs from a popular testing benchmark which we believe reasonably represent real-world programs in terms of code, test suite sizes and versions. Besides the correctness of data collection and correctness of the analysis code, the threats to internal validity concern the effectiveness of the algorithm for highly sparse matrices and affected tests.

We mitigate the first concern by using a pseudo-random procedure to generate several E matrices of varying sparsity to compare the estimated and computed values of number of clusters and their average sizes. Systematic methods to generate highly sparse matrices can be employed to further validate our results in this direction. To study the performance with respect to affected tests, we generated different sets of affected tests by randomly choosing the lines of interest that must be executed by these tests. Affected tests based on techniques described in [15] can be empirically studied to further validate our results.

6. Related Works

Amman and Knight [1] empirically observed that tests revealing the same fault often exhibit similar runtime behaviours. Clustering techniques provide one way to exploit this correspondence by generating clusterings in which tests with similar profiles are grouped into one cluster. These clusterings can be sampled in different ways to find faults, analyse a fault in detail to come up with comprehensive fixes and so on. Earlier works have applied clustering techniques to improve several testing aspects including observation-based testing [2]–[4], [16], regression test selection [6] and test suite minimization and prioritization [17], [7].

In general, the above applications of clustering have mostly employed Boolean valued function call test profiles and used some variants of the K-means clustering along with Euclidean distance. In [7], Boolean, line-based profiles are used, and the analysis is performed using agglomerative hierarchical clustering with the hamming distance. The number of clusters and the initial seeding of these clusters are manually chosen in all of these works. The C3 approach described in this paper is inspired by the cover-coefficient concept originally proposed by Can and Ozkarahan [10] and is significantly different from the earlier clustering approaches. The C3 approach is nonhierarchical, non-iterative, and uses a probabilistic notion of similarity between two tests that depend on all the available profiles and not just those of the two tests being compared (unlike Euclidean distance and its variants). The C3 approach also automatically determines the number of clusters and performs initial seeding of these clusters.

The approach in our paper is similar in spirit to observation-based testing where clustering techniques are used to selectively examine test outcomes after running them. Like observation-based testing we also assume that it is much more expensive to examine test outcomes than running them and therefore, it is crucial to judiciously choose the tests to be analysed. However, the proposed work is focused on regression testing, where the behaviours of a subset of available tests, the affected (or fault-revealing) tests, are of primary interest. Retrieving affected tests from a clustering is usually very different from retrieving an arbitrary group of tests as the distribution of affected tests over a clustering depends on a variety of factors including the impact of program changes on the test profiles, the noise due to the unaffected tests and so on. In this sense, our work may be viewed as extending the above works on observation-based testing to the regression testing context.

A unique aspect of our work is the comparison of multiple clusterings for performing cluster-based retrieval of affected tests. In most of the earlier works, only a single clustering has been considered for regression testing whereas as demonstrated by our experimental results, it may be worthwhile to compare which among the original and the new clustering structures is more appropriate to perform retrieval for regression testing. Another contribution of our work is a novel, empirical (non-safe) approach based on execution profiles to identify affected tests. To the best of our knowledge, this is the first application of the C3 approach to cluster test cases based on their execution profiles for testing individual and multiple program versions.

7. Conclusion

This paper developed a novel approach for cluster-based retrieval of tests for regression testing. A clustering approach based on the C3 was introduced to cluster tests based on their Boolean valued line profiles. Unlike typical clustering approaches, the number of clusters and the average size of clusters are automatically determined by the C3 approach without any additional user input. A novel and simple method was developed to identify the set of affected tests for regression testing based on the test profiles. The approach was applied to several programs (including several new and faulty versions) from the testing benchmark SIR [13]. Our results show that the retrieval of affected tests from an original clustering is likely to lead to better retrieval of affected tests in many cases compared to that using the newer clustering and hence must be considered before discarding the original clustering.

References

[1] P.E. Amman and J.C. Knight, Data diversity: An approach to software fault tolerance, *IEEE Transactions on Computers*, *37*(4), 1998, 418–425.

[2] W. Dickenson, D. Leon, and A. Podgurski, Finding failures by cluster analysis of execution profiles, *Proc. of International Conf. on Software Engineering*, 2001, 339–348.

[3] W. Dickenson, D. Leon, and A. Podgurski, Pursuing failure: The distribution of program failures in a profile space, *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2001.

[4] D. Leon, A. Podgurski, and L.J. White, Multivariate visualization in observation-based testing, *International Conf. on Software Engineering*, 2000.

[5] B. Guo, M. Subramaniam, and P. Chundi, Analysis of test clusters for regression testing, *International Workshop on Regression Testing, International Conf. on Software Testing (ICST)*, 2011, 736.

[6] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang, and B. Xu, An improved regression test selection technique by clustering execution profiles, *International Conf. on Quality Software*, 2010.

[7] S. Yoo, M. Harmon, P. Tonella, and A. Susi, Clustering test cases to achieve effective and scalable prioritization incorporating expert knowledge, *International Symposium on Software Testing and Analysis*, ACM, New York, 2009.

[8] V. Vangala, J. Czerwonka, and P. Talluri, Test case comparison and clustering using program profiles and static execution, *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2009.

[9] E. Wong and V. Debroy, A survey of software fault localization, *Technical Report, Department of Computer Science, University of Texas at Dallas, UTDSCS-45-09*, 2009.

[10] F. Can and E.A. Ozkarahan, Concepts and efffectiveness of the cover-coefficient-based methodology for text databases, *ACM Transactions on Database Systems*, *15*(4), 1990, 483–517.

[11] M.J. Harrold, G.Rothermel, R.Wu, and L.Yi, An empirical investigation of program spectra. *ACM Workshop on Program Analysis for Software Tools and Engineering*, 1998.

[12] A.K. Jain and R.C. Dubes, *Algorithms for clustering data* (Prentice Hall, 1988).

[13] H. Do, S. Elbaum, and G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, *Empirical Software Engineering*, *10*(4), 2005.

[14] S.B. Yao, Approximating block accesses in database organizations, *Communications of ACM*, *20*(4), 1977, 260–261.

[15] S. Yoo and M. Harman, Regression testing minimization, selection and prioritization: A survey, *Software Testing, Verification and Reliability*, 2010.

[16] A. Podgurski, W. Masri, Y. Mccleese, and F.G. Wolff, Estimation of software reliability by stratified sampling, *ACM Transactions on Software Engineering and Methodology*, *8*(3), 1999.

[17] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, Prioritizing test cases for regression testing, *IEEE Transactions on Software Engineering*, *27*(10), 2001.

Biographies

Mahadevan Subramaniam is an associate professor of computer science at the University of Nebraska-Omaha. He has a B.E. degree in computer science from the Birla Institute of Technology in India and a Ph.D. degree in computer science from the State University of New York at Albany. His research interests are formal methods, modelling and simulation and testing of hardware/software systems. He has significant industrial experience in architecting high-end hardware and system designs. He has published more than 70 conference and journal papers in leading IEEE, ACM conferences and international journals. He is the director of the Modeling and Simulation Lab at the Peter Kiewit Institute in the University of Nebraska-Omaha.

Parvathi Chundi is an associate professor in the computer science department at the University of Nebraska-Omaha. She has a B.E. degree in computer science from Anna University, India, and a Ph.D. degree in computer science from the State University of New York at Albany. She has extensive research experience in the areas of information retrieval, data mining and databases. She is the director of the Big Data Lab, holds five US patents and has published more than 50 research papers in journals such as the Springer's Data Mining and Knowledge Discovery Journal and conferences such as the SDM and CIKM. She also has several years of industrial research experience in research labs such as the HP Labs.