Building Test Oracles by Clustering Failures

Rafig Almaghairbe* and Marc Roper*
*Dept. Computer and Information Sciences
University of Strathclyde, Glasgow, UK
Email: rafig.almaghairbe,marc.roper@strath.ac.uk

***Abstract*—In recent years, software testing research has produced notable advances in the area of automated test data generation, but the corresponding oracle problem (a mechanism for determine the (in)correctness of an executed test case) is still a major problem. In this paper, we present a preliminary study which investigates the application of anomaly detection techniques (based on clustering) to automatically build an oracle using a system's input/output pairs, based on the hypothesis that failures will tend to group into small clusters. The fault detection capability of the approach is evaluated on two systems and the findings reveal that failing outputs do indeed tend to congregate in small clusters, suggesting that the approach is feasible and has the potential to reduce by an order of magnitude the numbers of outputs that would need to be manually examined following a test run.**

I. Introduction

Research in software testing has focused on automating many aspects of the testing process such as generating and executing test cases and maintaining and managing test suites. A relatively neglected, but essential, aspect of testing is the production of an oracle: a mechanism to determine the (in)correctness of an output associated with an input. Whilst there are tools capable of completely automatically generating test inputs [1], few techniques exist to generate test oracles, making the process of checking test outputs primarily human-centred and consequently expensive and error prone [2].

Existing approaches to generate oracles range from the inexpensive and ineffective to the effective but very costly. At one end of the scale, specified oracles can be generated from formal specifications [3], and are effective in identifying failures, but defining and maintaining such specifications is demanding and consequently such specifications are very rare. At the other end, implicit oracles are easy to obtain at practically no cost (e.g., the work of Carlos and Michael [4]) but are not able to identify semantic and complex failures, revealing only general errors like system crashes, null pointer dereferences or unhandled exceptions [3].

This paper reports on an empirical investigation into the use of clustering to build an automated oracle based on a system's input/output pairs with the aim of finding a technique which combines the effectiveness of a specified oracle and the cost of an implicit one. The key hypothesis behind this work is that normal data instances (passed tests) belong to large or dense clusters, while anomalies (failed tests) belong to small or sparse clusters. If this is true then the oracle problem may be reduced significantly to the task of examining just the outputs in the small clusters rather than all test outputs.

II. Background and Related Work

Two extensive reviews of testing oracles exist: one by Baresi and Young [2] and the other by Barr et al. [3] who classified the existing literature on test oracles into three broad categories: (1) specified oracles; (2) implicit oracles; and (3) derived oracles.

Specified oracles are obtained from a formal specification of the system behaviour. The ASTOOT tool, for instance, generates test suites along with test oracles from algebraic specifications [5]. Specified oracles are effective in finding failures but their success depends heavily on the availability of a formal specification: a limiting factor for most systems.

Implicit oracles are generated without reference to any domain knowledge or specification and are widely applicable. For example, the fuzzing approach proposed by Miller et al. [6] generates random inputs to a system with the aim of exposing weaknesses such as security vulnerabilities in the form of buffer overflows and memory leaks.

Derived oracles are created from properties of the system, artefacts other than the specification (e.g. documentation or execution information), or other versions of the system under test. For instance, metamorphic testing has been used to test search engines such as Google and Yahoo [7]. Our work is rooted in the area of oracles derived from system executions which can be subdivided in two main sections: oracles based on invariant detection and oracles based on anomaly detection.

A. Test Oracles Based on Invariant Detection

Invariants built into programs can be used to automatically check behaviour and hence form a type of test oracle. Invariants may be inserted into the code by developers but this can be a costly and an additional burden. To address this, various systems (examples include Daikon [8] and DIDUCE [9]) have been developed which aim to automatically derive invariants via dynamic analysis. These approaches have some commonality with the one presented in this paper but operate at white box rather than a black box (system) level.

B. Test Oracles Based on Anomaly Detection Techniques

The main principle behind creating test oracles using this approach is to identify items, events or observations which do not conform to an expected pattern and therefore may be indicative of faulty behaviour [10]. Existing work can be classified in to three main categories of learning technique: unsupervised, semi-supervised and supervised:

Unsupervised Learning Techniques do not require training data, and thus are most widely applicable. They make the implicit assumption that normal instances are far more frequent than anomalies in the test data. Examples of such work include that of Dickinson, Leon and Podgurski who demonstrated the advantage of automated clustering of function caller/callee execution profiles over random selection for finding failures [11], [12]. This has similarities with our approach but uses execution profiles rather than input/output pairs and furthermore is focused on reliability estimation rather than exploring software correctness. Yoo et al. also used a clustering approach to the problem of regression test optimisation [13] where test cases were clustered based on their dynamic runtime behaviour (execution traces).

Semi-Supervised Learning Techniques assume that training data has labelled instances for only the normal class (i.e. a subset of passing test cases needs to be identified). A model is built for the class that corresponds to normal behaviour which is then used to identify anomalies in the subsequent data. Examples include the work of Podgurski et al. on bug clustering for the purposes of fault localisation [14], and of Bowring and colleagues on reverse engineering [15].

Supervised Learning Techniques assume the availability of a training data set which has labelled instances for normal as well as anomaly classes and is therefore the least generally applicable. This has been employed in regression testing where a reference version of the software which makes for accurate data labelling [16] and explored in the image processing domain [17].

From the above it can be seen that there is a body of work that explores the use of anomaly detection strategies to support software testing, but these typically operate on quite different types of data, or utilize semi-supervised or supervised learning strategies, and the application to input/output pairs has not been extensively investigated.

III. Cluster Analysis

Clustering aims to partition a population of objects, each containing various attributes, into groups in such way that objects with similar attribute values are placed in the same cluster, whereas those with dissimilar ones are placed in different clusters [18]. The similarity of objects can be decided by using different distance metrics and there are a large variety of approaches towards clustering. So far this work has only explored the use of the technique known as hierarchical clustering – a stepwise process which can be divided into two methods: agglomerative and divisive. The agglomerative method initially assigns each object to its own cluster, calculates the distance between two clusters, and combines the most similar ones. This process is repeated until no close similarity or dissimilarity between two clusters can be found. The divisive method, on the other hand, initially assigns all objects into one cluster and then divides this main cluster into smaller clusters based on object dissimilarity until no further splits can be made.

IV. Experimental Investigation

With the aim of exploring the main hypothesis behind this work (normal data instances belong to large and dense clusters, while anomalies either belong to small or sparse clusters), and to understand the potential of this approach as an effective oracle mechanism, a series of experiments was run to evaluate the effectiveness of clustering techniques applied to input/output pairs for isolating failures.

A. Subject Programs

Two medium size Java systems were used as subject programs: NanoXML and Siena. NanoXML is a non-GUI based XML parser written in Java and available from the Software Infrastructure Repository (SIR)[1] which has 24 classes, 5 versions containing multiple faults, and 214 test cases. The error rates in all faulty versions ranged from 31% to 39%. The fourth version was excluded as it contains no faults.

Siena (Scalable Internet Event Notification Architecture) is an event notification middleware, also available from the SIR containing 26 classes (9 in its core and 17 which constitute an application), 567 test cases and 7 faulty versions: 3 with a single fault, and 4 with multiple ones. The first faulty version was excluded from the experiments because the outputs are indistinguishable from the original. The error rate in all remaining versions was 17%.

Both systems also come with test suites – an important factor in choosing these systems as having sets of good, but independently created, tests is vital for this experiment.

B. Experimental Protocol

The basic principle of the experiment involved taking each of the systems (original and faulty versions), running them on the provided inputs to produce the outputs, and applying the clustering algorithms to this resultant set of input/output pairs. Since the failures associated with the faults are known, the effectiveness of the clustering approach can then be evaluated. This process is described in more detail below:

TABLE I: Example Coding of Input/Output Pairs

	Input	Output
Nanoxml	**Flower colour="Red" smell="Sweet" name="Rose" season="Spring"**	xml element name is: **Flower**
Encoding	**FCRSSNRSS**	**F**
Siena	Filter senp{x=0}**filter**{**x=20 y=30 z=10**} Event senp{x=0}**event**{**x=20**} senp{x=0}**event**{**y=30 z=10**}	**subscribing** for **filter**{**x=20 y=30 z=10**}**publishing** for **event**{**x=20**}**publishing** for **event** {**y=30 z=10**}
Encoding	**F111E1E11**	**SF111PE1PE11**

[1]http://sir.unl.edu/content/sir.php

1) Executing Test Cases: Both subject programs come with Test Specification Language (TSL) test suites and tools to run these automatically (details are available from the SIR repository and the article by Do, Elbaum and Rothermel [19]). Test cases which failed to produce any output were discarded (7 out of 214 for Nanoxml, and 73 out of 567 for Siena giving final test case numbers of 207 and 494).

2) Input/Output Pair Transformation: Before feeding the input/output pairs to the clustering algorithm the data was transformed from text to an attribute of vectors by a simple process of tokenisation [18]. Table I shows an example of this for both NanoXML and Siena. Notice that the parameters for the Sience commands were all encoded as “1” as they remained unchanged between input and output.

3) Identify Failures: The NanoXML system comes with matrices which map test cases to faults and makes the identification of faults effectively automatic. Siena has no such fault matrix so the test outputs of the original version were compared with that of the faulty ones to find the failing tests.

4) Perform Clustering: Agglomerative hierarchical clustering is used in all of the experiments. This type of clustering was chosen because it performed reasonably well for some similar problems [11], [12], [20], [13] and was also suggested by Witten et al. [18] as the most suitable solution for nominal and string data (which the coding system produces for these two systems). The inputs to cluster analysis were the coded input/output pairs of the subject programs.

After exploring various alternatives Euclidean distance was settled on as the measure of (dis)similarity between two objects. The WEKA toolkit[2] used in this study computes this by taking the nominal data attributes and transforming them into binary variables. The squared differences between the binary vectors are then summed: a zero sum indicates agreement (similarity), but a non-zero sum suggests a dissimilarity.

In addition to a similarity metric, clustering requires a linkage metric which is used to determine when clusters should be merged or split. There are three approaches: *Single Linkage* calculates the minimum distance between an object in one cluster and an object in another, *Average Linkage* computes the mean distance between objects in the two clusters, and *Complete Linkage* is based on the maximum distance between objects. All three were used in this study but for space reasons only the average linkage results are reported.

5) Number of Clusters: For the clustering approach adopted the number of clusters needs to be provided as a parameter. This can clearly have a significant impact – too many clusters results in fragmentation and too few in over-generalisation. Therefore, a number of different cluster counts were explored based on a percentage of the number of subject program test cases (1%, 5%, 10%, 15%, 20% and 25%).

V. Experimental Results And Discussion

A. Distribution of Failures

The first major question to explore is whether failures are distributed in a random pattern. Figures 1 and 2 show bar charts of NanoXML (faulty version 3) and Siena (faulty version 2) systems with 50 and 100 clusters respectively. It can be seen from these that failures in the input/output pairs population tend to cluster together and these clusters tend to be the smaller ones (these are selected results but others reflect a similar pattern). It would seem that a substantial number of failures belong to small clusters, supporting the main hypothesis behind this work.

[2]http://www.cs.waikato.ac.nz/ml/weka/downloading.html

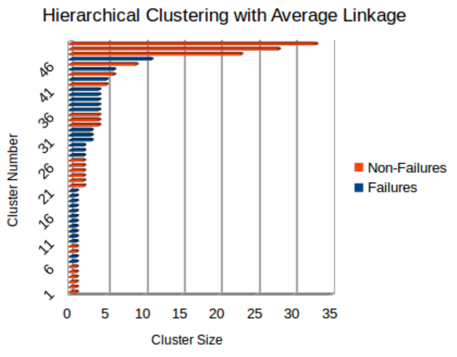

Fig. 1: Hierarchical Clustering Algorithm with Average Linkage for Nanoxml (Version 3)

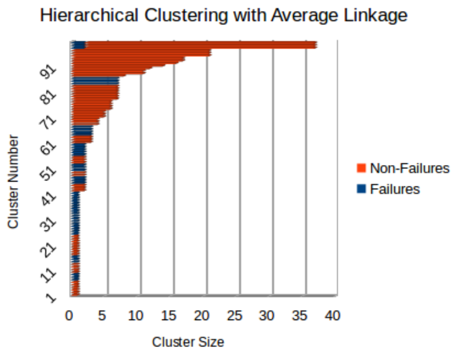

Fig. 2: Hierarchical Clustering Algorithm with Average Linkage for Siena (Version 2)

B. Failures Found Verses Cluster Counts and Cluster Sizes

TABLE II: Percentage of Failures vs. Cluster Size

Cluster	Cluster	NanoXML				Cluster	Siena
Count (%)	Size (%)	V1	V2	V3	V5	Size (%)	(ave.)
1	50	7.40	2.81	**100**	10.76	19.8	0
5	10	56.79	**63.38**	34.28	26.15	4	16.66
10	6.25	56.79	**63.38**	45.71	**61.53**	2	41.66
15	3.25	56.79	**63.38**	**82.85**	52.30	1.21	41.66
20	2.50	51.85	54.92	75.71	52.30	0.79	67.85
25	2.25	**65.43**	61.97	75.71	**61.53**	0.6	**75**

To explore this finding further we examined the population of input/output pairs that were in small clusters (defined as being of average size or less) and corresponded to failures.

Table II shows, for varying numbers and sizes of clusters over both systems, the percentage of all data points corresponding to failures. The first column (Cluster Count %) defines the number of clusters the algorithm is charged with creating expressed as a percentage of the number of test cases. The second column (Cluster Size %) is the average size of the clusters again in terms of the number of tests. The subsequent columns refer to the version number of the program, following which is the same information for Siena.

The data shows that when the cluster counts are between 15% to 25% of the number of test cases (corresponding to cluster sizes of around 3% of the number of test cases), well over 60% of the data points are failures, lending strong support to the main hypothesis of this paper. One case where this is not quite true is version 3 of NanoXML where the largest clusters contained the most failures: the input-output pairs corresponding to failures are so distinct from the rest that they were all grouped into one cluster (an impressive but probably unusual case!).

The trend is for the failure density of the clusters to increase in line with the number of clusters (with the exception of the aforementioned version 3 in NanoXML). However, in some cases, as the number of clusters increases the failure intensity peaks and then begins to drop (although not substantially) as the clusters are forced to fragment. An important lesson from this is not to create too many clusters in relation to the number of input-output pairs. This phenomenon is less pronounced in the Siena data. Note that the faults in Siena changed the same output data in all versions which explains why there is no separation of the results into versions.

C. Failure Density of Smallest Clusters

From the perspective of supporting the construction of a test oracle, the interesting question concerns the return on investment: how many outputs need to be examined before a reasonable number of failures are observed? To answer this we examined in more detail the proportion of failing outputs appearing in the smallest sized clusters. The absence of a fault matrix for Siena makes this very time consuming to compute, therefore only the results for the highest failure density clusters for NanoXML have been calculated so far. The results of this are summarised in Table III and show the cluster size (the 3 values correspond to the absolute size of the cluster, the number of clusters of that size, and the size of the cluster and proportional to the test set size) and details of the failures found (the proportion, the actual failures indicated by ‘Fn’, and the number of occurrences of each failure). So, for instance, the first entry of Table III shows that for Version 1 using 25% of the number of test cases to define the number of clusters, there were 13 clusters each of size 1 corresponding to 0.48% of the number of test cases, containing failures 1 (3 times), 2 and 6 (once each).

The table shows that on average over all four versions a fair proportion of the failures - 45% (13/29) - are contained within the very smallest clusters (formed from just one or two items). This is encouraging from an oracle perspective: out of 43 outputs, 23 correspond to failures giving a failure density of 53%. This initially good rate tails off until the cluster size reaches 4 and additional failures appear in the outputs (except for version 5). By this point an average of 66% (19/29) of the failures have appeared in the clusters, albeit at the expense of having to examine more non-failing outputs and encountering duplicate failing outputs (but still giving a failure density of around 59%). This failure density figure, combined with the fact that clusters tend to contain outputs associated with the same failure, means that in practice less than half of the outputs from a cluster need to be checked before a failing output is encountered.

TABLE III: Failure Distribution over less than Average Sized Clusters

Version 1 (25%)	
Cluster Size	Failures
1, 13, 0.48%	3/7 (F1:3,F2:1,F6:1)
2, 13, 0.97%	3/7 (F1:4,F2:2,F6:2)
3, 4, 1.45%	1/7 (F6:3)
4, 8, 1.94%	3/7 (F2:16,F5:8,F7:8)

Version 2 (15%)	
Cluster Size	Failures
1, 7, 0.48%	3/7 (F1:3,F2:1,F6:1)
2, 3, 0.97%	1/7 (F6:2)
3, 5, 1.45%	1/7 (F6:3)
4, 6, 1.94%	3/7 (F2:8,F5:8,F7:8)
5, 2, 2.42%	1/7 (F2:5)
6, 1, 2.91%	(1/7) (F2:6)

Version 3 (15%)	
Cluster Size	Failures
1, 10, 0.48%	5/7 (F1:4,F2:1,F3:1,F4:2,F6:1)
2, 4, 0.97%	3/7 (F1:1,F4:2,F6:2)
3, 5, 1.45%	2/7 (F4:6,F6:3)
4, 6, 1.94%	3/7 (F2:8,F5:8,F7:8)
5, 2, 2.42%	1/7 (F2:5)
6, 1, 2.91%	1/7 (F2:6)

Version 5 (25%)	
Cluster Size	Failures
1, 13, 0.48%	2/8 (F1:3,F2:1)
2, 14, 0.97%	2/8 (F1:2,F2:2)
3, 8, 1.45%	1/8 (F2:3)
4, 7, 1.94%	1/8 (F2:28)

Of course, there are still additional failing outputs embedded in the larger clusters which can’t be ignored. This is clearly a weakness of the approach and one of the main topics of future work is to explore how these can be teased out into smaller clusters. A further feature of the clustering is that there is often number of independent clusters associated with the same failure (separated typically because the input/output pairs have different attribute values). This is also a challenge since finding the same failure appearing in several clusters can be quite frustrating for the individual charged with the task of checking outputs. Merging them together is not the answer as this will typically result in a larger cluster which may escape scrutiny, so some way of indicating similarity between them needs to be explored.

D. Threats to Validity

The main threat to the validity of this study is the limited number and types of subject programs used in our experiments along with their associated faults and failure rates.

The input/output pairs of all subject programs were string data (any numeric values were transformed to strings), and all subject programs were moderate size. The coding scheme also indicates a potential threat but this was created by examining a subset of inputs and outputs in ignorance of whether they are passing or failing pairs, and then applied automatically to the remainder of the data set. This is relatively early work in this area and the aim is to mitigate these threats by exploring a wider range of systems in the near future.

VI. Conclusion

This paper presents an initial, and we believe the first, investigation study into building an automated test oracle by using hierarchical clustering with input/output pairs alone. The approach was evaluated on several versions of two modest-sized Java systems using supplied sets of test data and faults, along with hierarchical clustering and varying sizes of cluster. The results suggest that over 60% of the contents of small (average-sized or less) clusters correspond to failures and furthermore these smallest clusters are dominated by failing outputs – which supports the experimental hypothesis behind this work. This result has potentially important practical consequences: the task of scrutinising test outputs may potentially be reduced significantly to examining well under half the contents of the smaller clusters an order of magnitude reduction in effort.

Although the results are encouraging it is clearly not reasonable to generalise from this small set of experiments and further empirical evaluation is needed to confirm our results. The two main challenges are firstly to improve the count of unique failures in the smaller clusters by exploring how the input/output pairs may be augmented with additional data such as trace executions, timing information, code coverage information and profile executions; and secondly to examine how similar failures in unique clusters can be flagged without inadvertently grouping them together into larger clusters. Additional work includes exploring other anomaly detection and clustering strategies with the aim of reducing further the number of outputs that need to be considered, and evaluating the approach against existing strategies from the specification mining domain

The complete results and data sets used in the study may be found at: http://personal.strath.ac.uk/rafig.almaghairbe/

References

[1] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

[2] L. Baresi and M. Young, "Test oracles," Tech. Rep., 2001.

[3] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *Software Engineering, IEEE Transactions*, to appear.

[4] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[5] R.-K. Doong and P. G. Frankl, "The astoot approach to testing object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 2, pp. 101–130, Apr. 1994. [Online]. Available: http://doi.acm.org/10.1145/192218.192221

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[7] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Softw. Test. Verif. Reliab.*, vol. 22, no. 4, pp. 221–243, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.437

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.01.015

[9] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 291–301. [Online]. Available: http://doi.acm.org/10.1145/581339.581377

[10] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

[11] W. Dickinson, D. Leon, and A. Fodgurski, "Finding failures by cluster analysis of execution profiles," in *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, May 2001, pp. 339–348.

[12] W. Dickinson, D. Leon, and A. Podgurski, "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*, ser. ESEC/FSE-9. New York, NY, USA: ACM, 2001, pp. 246–255. [Online]. Available: http://doi.acm.org/10.1145/503209.503243

[13] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM Press, July 2009, pp. 201–211.

[14] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 465–475. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776872

[15] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 195–205. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007539

[16] M. Vanmali, M. Last, and A. Kandel, "Using a neural network in the software testing process," *International Journal of Intelligent Systems*, vol. 17, no. 1, pp. 45–62, 2002. [Online]. Available: http://dx.doi.org/10.1002/int.1002

[17] L. Briand, "Novel applications of machine learning in software testing," in *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, Aug 2008, pp. 3–10.

[18] I. H. Witten and E. Frank, "Data mining: Practical machine learning tools and techniques," 2005.

[19] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 405–435, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10664-005-3861-2

[20] S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou, "A dynamic test cluster sampling strategy by leveraging execution spectra information," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, April 2010, pp. 147–154.