2010 34th Annual IEEE Computer Software and Applications Conference Workshops

Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing

Zhi Quan Zhou
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, NSW 2522, Australia
Email: zhiquan@uow.edu.au

***Abstract*—Random Testing (RT) is a fundamental software testing technique. Adaptive Random Testing (ART) improves the fault-detection capability of RT by employing the location information of previously executed test cases. Compared with RT, test cases generated in ART are more evenly spread across the input domain. ART has conventionally been applied to programs that have only numerical input types, because the distance between numerical inputs is readily measurable.**

The vast majority of computer programs, however, involve non-numerical inputs. To apply ART to these programs requires the development of effective new distance measures. Different from those measures that focus on the concrete values of program inputs, in this paper we propose a method to measure the distance using coverage information. The proposed method enables ART to be applied to all kinds of programs regardless of their input types.

Empirical studies are further conducted for the branch coverage Manhattan distance measure using the *replace* and *space* programs. Experimental results show that, compared with RT, the proposed method significantly reduces the number of test cases required to detect the first failure. This method can be directly applied to prioritize regression test cases, and can also be incorporated into code-based and model-based test case generation tools.

***Keywords*-software testing; adaptive random testing; adaptive random sequence; test case prioritization; distance measure.**

I. Introduction

Testing is a critical activity in software development and evolution, and accounts for over fifty percent of the development cost in a typical commercial organization [1], [2]. Testing, however, is an imperfect process and is always conducted under time and resource constraints. It is therefore important to know how to perform testing in a more effective way at a lower cost.

Random Testing (RT) is a fundamental testing method [3], [4]. As pointed out by Chen et al. [5], [6], RT is simple in concept, often easy to implement, has been demonstrated to be effective in detecting failures, and is good at exercising systems in a way that may not be expected by human testers. When the source code and the specifications of the program under test are not available or incomplete, RT may be the only practical choice. RT has been popularly applied to test real-world software for decades, and forms a core part of many other testing methods [2], [7], [8], [9], [10].

On the other hand, since RT does not use any available information to guide test case selection, it is often argued that RT is inefficient. To reduce the number of test cases required to detect a failure, Malaiya [11] introduced an *antirandom testing* method, in which the first test case is selected randomly, and each subsequent test case is selected by choosing the one whose total distance to all the previously executed test cases is maximum. Antirandom testing has a limited degree of randomness: the first test case is selected randomly; whereas the sequence of all the subsequent test cases is deterministic. In antirandom testing, the total number of test cases also needs to be known in advance.

An *Adaptive Random Testing* (ART) method has been examined by a growing body of research [5], [6], [12], [13], [14], [15], [16], [17]. As noted by Jaygarl et al. [18], ART "became one of the most effective approaches in the automatic test generation area." ART improves the fault-detection capability of RT in terms of using fewer test cases to detect the first failure. ART is based on the observations that many program faults result in failures in contiguous regions of the input domain (in other words, failure-causing inputs are often clustered). In such situations, if previously executed test cases have not revealed a failure then, intuitively, selecting an input close to them will also be less likely to detect a failure. ART therefore systematically guides the generation or selection of random test cases by making them evenly spread over the input domain. Based on this principle, several ART algorithms (or methods) have been proposed, all of which make use of the location information of previously executed test cases that have not revealed a failure. Since the location information and pass/fail information is a kind of feedback from previous tests, it is reasonable to consider ART an instance of *software cybernetics* [19]. In general, software cybernetics explores the interplay between software and control. It should be noted that ART differs from antirandom testing in that ART preserves the randomness of RT and that ART does not require the predetermination of the total number of test cases.

ART has conventionally been applied to programs that have only numerical input types because it calculates the distance between test cases using the Euclidean measure. Merkel [20] and Kuo [21] proposed a distance measure

978-0-7695-4105-1/10 $26.00 © 2010 IEEE
DOI 10.1109/COMPSACW.2010.43

based on the concepts of categories and choices proposed by Ostrand and Balcer [22] for the category-partition method. Ciupa et al. [16] applied ART to object-oriented software by proposing a new measure for calculating the distance between objects, and this work was further enhanced by Jaygarl et al. [18] for the purpose of testing object-oriented software. Chen et al. [6] presented a synthesis of significant research results and insights into ART. In particular, Chen et al. pointed out the potential of using ART to order a given test suite, such as for regression test case prioritization. Such an order is called an *adaptive random sequence*.

In this paper we propose a method to measure the distance [1] between test cases based on coverage information, and conduct empirical evaluation to investigate the effectiveness of the proposed method. This method enables ART to be applied to any types of programs without any requirements on their input types. Specifications of the program under test are not required either.

The rest of this paper is organized as follows: Section II introduces an ART algorithm, namely FSCS-ART, that we will adopt for this study. Section III proposes a coverage-based distance measure for use with the FSCS-ART algorithm. Section IV describes the design of experiments for empirical evaluation, and Section V presents the experimental results. Section VI briefly reviews some related work. Section VII discusses related issues, points out future research directions, and concludes the paper.

II. The ART Algorithm

Following the previous practices [13], [17], *F-measure* is adopted as a metric to compare the fault-detection capabilities of different testing methods. F-measure refers to the number of test cases needed to be run to reveal the first failure.

The principle of ART is to evenly spread test cases. This principle can be implemented in different ways and, therefore, several ART algorithms have been developed. The first ART algorithm proposed is known as the *Fixed Size Candidate Set ART* (FSCS-ART) [12], [14]. In this algorithm, an initial test case is randomly chosen and run. Then, to choose a new test case, a fixed number of candidates are randomly generated (the recommended number of candidates is 10 [12], [14]). For each candidate c_i, the closest previously executed test case is located and the distance d_i is recorded. In other words, d_i is the closest distance between c_i and the set of all the previously executed test cases. The candidate with the maximum d_i is selected as the next test case, and all the other candidates are discarded. This process is repeated until the testing stopping criterion is met. The time complexity of this algorithm is in $O(n^2)$, where n is the total number of test cases finally generated.

[1] In this paper, the terms *distance measure* and *difference measure* are used interchangeably.

The FSCS-ART algorithm is adopted in this study because of its simplicity. The number of candidates has been set to 10 as recommended.

III. The Difference Measure

Consider the case of *regression testing*. Situations other than regression testing will be discussed in Section VII. Normally, in regression testing, certain operational profile data of previous tests are available. For example, Rothermel et al. [23] studied nine different test case prioritization techniques. Apart from the first three techniques (namely "no prioritization", "random", and "optimal") that served only as experimental controls, all the other six techniques use test coverage information, such as statement and branch coverage, produced by prior executions of test cases, to prioritize the same set of test cases for subsequent execution.

Following the same source of motivation that "the availability of test execution data can be an asset" [23], in this paper we propose a metric, namely the *Coverage Manhattan Distance* (CMD), to measure the difference between any two arbitrary test cases. During the execution of a program, we can observe whether certain elements of the program have been touched/covered. Such an element can be, for instance, a node or an edge in the program's control flow graph, data flow graph, call graph, or other types of graphs/diagrams. More concrete examples of such elements include, to name a few, statements, branches, basic blocks, functions, and function calls. To record coverage information, if an element of the given type has been exercised at least once, then its corresponding flag is set to 1; otherwise the flag is set to 0. Note that CMD concerns coverage but does not count the frequency. In other words, frequencies 3 and 100 are both treated as "1".

In this paper we focus on *branch coverage*. Let x be a test case, and E_x be a vector that records the branch coverage information of x. More formally, let $E_x = (x_1, x_2, \ldots, x_n)$ be an execution profile of x, where $x_i \in \{0, 1\}$ for $i = 1, 2, \ldots, n$, and n is the total number of branches in the program (note that each condition has two branches, namely a *true* branch and a *false* branch). The value of x_i is 1 if and only if the ith branch of the program has been covered during the execution of x; otherwise $x_i = 0$. Similarly, let y be another test case, and $E_y = (y_1, y_2, \ldots, y_n)$ records the branch coverage information of y. The CMD between x and y is given by

$$CMD(x, y) = \sum_{i=1}^{n} |x_i - y_i|. \quad (1)$$

In the context of regression testing, E_x and E_y actually store coverage data of the previous tests. Therefore, here we have an assumption that past coverage data are useful to predict subsequent execution behavior with sufficient accuracy after modifications are made to the program code. This assumption is not only supported by the literature of

regression testing, but also supported by our empirical study results to be reported shortly.

IV. The Experiments

To assess the fault-detection capability of the proposed method, two well-known subject programs were used to conduct empirical evaluation. The first one is the *replace* program of the *Siemens suite of programs* [24], downloaded from http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/. The *replace* program performs regular expression matching and substitutions. It is the largest and most complex one among the Siemens suite of programs, with 512 lines of C code (excluding blanks and comments) and 20 functions. Also included in the *replace* package are 32 faulty versions that cover a variety of logic errors and 5,542 test cases.

The second subject program is the *space* program, downloaded from the *Software-artifact Infrastructure Repository* (http://sir.unl.edu) [25]. The program consists of 6,199 lines of C code (excluding blanks and comments) and 136 functions, and works as an interpreter for an array definition language. The *space* package includes 38 faulty versions which, according to the Software-artifact Infrastructure Repository, were real faults discovered during the program's development. Also included in the *space* package are 13,551 test cases.

As introduced above, each subject program package includes a base version, associated faulty versions, and a suite of test cases. For each subject program, the experiment was conducted as follows: we first run the base version using all the provided test cases. During each test case execution of the base version, the Linux utility *gcov* (a standard test coverage tool in concert with *gcc*) was used to collect branch coverage data. Outputs of the base version were also recorded as the test oracle.

Then, for each faulty version, we generated two sequences (permutations) of all the test cases: a pure random (PR) sequence and an adaptive random (AR) sequence. The latter was generated using the FSCS-ART algorithm with the branch coverage Manhattan distance as the difference measure between test cases. Note that the coverage data were collected from the base version rather than the faulty version. The F-measures of the PR sequence and the AR sequence were recorded (a failure is detected when the faulty version's output differs from the base version's output). For each faulty version, this process was repeated 1,000 times.

V. Experimental Results

A. Results of Experiments with the Replace Program

Results of experiments with the *replace* program are shown in Table I . The *replace* package includes 32 faulty versions. Version 32 is excluded from the experiments because it generated identical outputs as the base version on all the 5,542 test cases. Furthermore, versions 13, 23, and 26 are not stable as each of them produced different outputs (hence different sets of failure-causing test cases) when run at different times or under different environments. We therefore also excluded these versions from the experiments.

Table I
Results of experiments with *replace* program (1,000 trials, 5,542 test cases). OFRT: Observed mean F-measure of RT; TFRT: Theoretical mean F-measure of RT; FART: observed mean F-measure of ART.

	OFRT	TFRT	FART	FART/OFRT	FART/TFRT
v1	80.017	80.333	37.109	46.38%	46.19%
v2	146.697	145.868	83.123	56.66%	56.99%
v3	43.680	42.313	11.867	27.17%	28.05%
v4	38.316	38.493	11.497	30.01%	29.87%
v5	20.381	20.379	14.817	72.70%	72.71%
v6	56.116	57.144	59.850	106.65%	104.73%
v7	66.081	65.988	26.754	40.49%	40.54%
v8	96.982	100.782	42.675	44.00%	42.34%
v9	187.797	178.806	163.446	87.03%	91.41%
v10	230.119	230.958	168.703	73.31%	73.04%
v11	169.558	178.806	165.801	97.78%	92.73%
v12	18.221	17.881	15.759	86.49%	88.13%
v14	41.706	40.167	22.755	54.56%	56.65%
v15	92.387	90.869	74.041	80.14%	81.48%
v16	63.952	65.988	25.656	40.12%	38.88%
v17	220.298	221.720	222.882	101.17%	100.52%
v18	25.546	26.270	20.143	78.85%	76.68%
v19	1348.250	1385.750	557.812	41.37%	40.25%
v20	238.456	241.000	245.617	103.00%	101.92%
v21	1381.730	1385.750	777.449	56.27%	56.10%
v22	270.818	277.150	256.119	94.57%	92.41%
v24	30.952	32.415	12.369	39.96%	38.16%
v25	1360.870	1385.750	752.841	55.32%	54.33%
v27	20.309	20.996	8.392	41.32%	39.97%
v28	38.684	38.762	11.689	30.22%	30.16%
v29	83.424	85.277	25.591	30.68%	30.01%
v30	20.150	19.449	6.719	33.34%	34.55%
v31	26.172	26.270	18.495	70.67%	70.40%
avg	229.202	232.191	137.142	col:61.44% row:59.83%	col:61.04% row:59.06%

In Table I , the column OFRT gives the observed mean F-measure of random testing, calculated from the 1,000 trials. The column TFRT gives the theoretical mean F-measure of random testing. Let n be the total number of test cases and m ($0 < m \leq n$) be the total number of failure-causing test cases. Because the testing is through sampling *without* replacement, we cannot calculate TFRT by simply calculating n/m. Instead, TFRT is obtained as follows:

$$TFRT = \sum_{i=1}^{n-m+1} p_i \times i, \qquad (2)$$

where p_i is the probability that the first failure is detected

at the ith test run. p_i is calculated iteratively as follows:
$p_1 = m/n$;
$p_2 = (1 - p_1) \times (m/(n-1))$;
$p_3 = (1 - p_1 - p_2) \times (m/(n-2))$;
...
$p_{n-m+1} = (1-p_1-\ldots-p_{n-m}) \times (m/(n-(n-m+1-1)))$
$= (1 - p_1 - \ldots - p_{n-m})$.

For the *replace* program, $n = 5,542$, and different faulty versions have different m values. The column FART gives the observed mean F-measure of adaptive random testing, calculated from the 1,000 trials. The columns FART/OFRT and FART/TFRT give the ratio of ART F-measure to RT F-measure. A smaller ratio indicates a better fault-detection capability of ART, and a ratio smaller than 1 means ART outperformed RT.

Each row of Table I corresponds to a faulty version, and the last row shows the average of each column. Note that column 5 of the last row shows two values: "col:61.44%" is the mean of the column, whereas "row:59.83%" is the mean FART (137.142) divided by mean OFRT (229.202). Column 6 of the last row is explained similarly.

Table I shows that ART outperformed RT in all the versions except v6, v17, and v20 (highlighted), but the differences between ART and RT in these three versions are marginal. The highest saving occurs for v3, for which the ratio is around 28%, which means that ART used about 72% fewer test cases than RT to detect the first failure. Overall, the average ratio of ART F-measure to RT F-measure is around 61%, which clearly indicates a significant saving.

It is worth noting that, in Table I , no correlation could be found between the fault-detection capability of ART (in terms of FART/OFRT or FART/TFRT ratios) and the failure rate (a higher TFRT indicates a smaller failure rate). This observation does not agree with the simulation results reported in Chen et al. [5], which state that ART will achieve more gain for smaller failure rates. We note, however, that the experiments conducted in this study have a different context (and probably different failure patterns) from the numerical simulations conducted in Chen et al. [5]. Further investigation is needed to understand this phenomenon.

B. Results of Experiments with the Space Program

Experimental results with the *space* program are shown in Table II , where versions 1, 2, 32, and 34 are excluded from the experiments because they produced identical outputs as the base version. Table II shows that ART outperformed RT in all versions except v3. Note that the TFRT values for v4, v6, and v30 are smaller than 2, which indicates a very high failure rate in these versions. Given that the smallest possible F-measure is 1, there is little room for improvement for these versions. The minimum ratio of ART F-measure to RT F-measure is achieved for v22 (below 17%).

The average ratio of ART F-measure to RT F-measure is between 30.92% and 54.78%, as shown in the last two columns of the last row. This result is even better than that of the *replace* program. We reckon that ART should perform better for larger programs, the inputs of which normally have a greater diversity in coverage. While this work cannot be directly compared with Chen and Merkel [17], we note that theoretically no testing method can achieve an F-measure less than half of the F-measure of random testing with replacement, when the size, shape, and orientation (but not the location) of failure regions are known [17].

Table II
RESULTS OF EXPERIMENTS WITH *space* PROGRAM (1,000 TRIALS, 13,551 TEST CASES). OFRT: OBSERVED MEAN F-MEASURE OF RT; TFRT: THEORETICAL MEAN F-MEASURE OF RT; FART: OBSERVED MEAN F-MEASURE OF ART.

	OFRT	TFRT	FART	FART/OFRT	FART/TFRT
v3	20.460	20.978	23.221	113.49%	110.69%
v4	1.099	1.092	1.085	98.73%	99.36%
v5	3.297	3.419	2.957	89.69%	86.49%
v6	1.060	1.066	1.049	98.96%	98.38%
v7	84.664	82.634	22.786	26.91%	27.57%
v8	146.672	141.167	47.787	32.58%	33.85%
v9	3.341	3.163	2.641	79.05%	83.51%
v10	11.131	11.163	6.718	60.35%	60.18%
v11	12.641	12.607	8.212	64.96%	65.14%
v12	406.105	398.588	124.465	30.65%	31.23%
v13	17.424	17.487	8.024	46.05%	45.89%
v14	7.604	7.691	4.034	53.05%	52.45%
v15	3.931	3.879	2.435	61.94%	62.78%
v16	27.261	26.889	6.898	25.30%	25.65%
v17	70.237	68.792	68.233	97.15%	99.19%
v18	393.625	398.588	124.836	31.71%	31.32%
v19	11.015	10.798	8.594	78.02%	79.59%
v20	65.505	64.228	18.786	28.68%	29.25%
v21	64.621	64.228	18.528	28.67%	28.85%
v22	209.063	208.492	34.768	16.63%	16.68%
v23	48.229	49.280	16.954	35.15%	34.40%
v24	19.157	18.413	6.032	31.49%	32.76%
v25	3.181	3.118	2.594	81.55%	83.21%
v26	8.197	8.179	4.133	50.42%	50.53%
v27	388.891	387.200	66.494	17.10%	17.17%
v28	1.932	2.003	1.683	87.11%	84.01%
v29	19.462	18.667	9.539	49.01%	51.10%
v30	1.262	1.256	1.193	94.53%	95.01%
v31	8.436	8.263	5.119	60.68%	61.95%
v33	398.249	410.667	74.055	18.60%	18.03%
v35	63.649	63.624	19.210	30.18%	30.19%
v36	145.472	148.923	129.137	88.77%	86.71%
v37	145.906	145.720	43.199	29.61%	29.65%
v38	382.884	366.270	72.613	18.96%	19.82%
avg	93.990	93.486	29.059	col:54.58% row:30.92%	col:54.78% row:31.08%

VI. Related Work

Very recently, Jiang et al. [26] used the *Jaccard distance* to measure the difference between two test cases for ART. The Jaccard distance between two test cases a and b is given by $D(a,\ b) = 1 - |A \cap B|/|A \cup B|$, where A and B are the sets of statements (or branches or functions) covered by a and b, respectively. Furthermore, instead of using a fixed number of candidates, Jiang et al.'s algorithm constructs the candidate set by iteratively adding a random candidate into the set. This construction procedure will stop when the new candidate cannot increase program coverage or the candidate set is full. Jiang et al. empirically investigated the performance of a family of ART algorithms that employ the Jaccard distance for regression test case prioritization. Their empirical study results show that ART was statistically superior to RT in terms of the *APFD* metrics and that one of the ART algorithms was consistently comparable to the "additional" algorithm (one of the best coverage-based regression test case prioritization algorithms) in terms of APFD, and yet had a lower time cost.

Jiang et al. [26] and the author of the present paper have conducted their studies independently without knowledge of each other's work. A comparison of Manhattan-distance-based ART and Jaccard-distance-based ART against the APFD metrics is, therefore, left as immediate future work. An obvious difference between the two is that whenever the intersection of sets A and B is empty, the Jaccard distance between A and B always takes the maximum value "1", but this is not the case with the Manhattan distance. For instance, suppose there are five branches in the program under test, namely b_1, b_2, ..., b_5. Consider three test cases x, y and z, of which the branch-coverage execution profiles are $(1,\ 0,\ 0,\ 0,\ 0)$, $(0,\ 1,\ 0,\ 0,\ 0)$ and $(0,\ 1,\ 1,\ 1,\ 1)$, respectively. Because of the empty intersection of the covered branches, the Jaccard distance between test cases x and y is 1, and that between x and z is also 1. In comparison, the Manhattan distance between x and y is 2, whereas that between x and z is 5. As a result, if $\{x\}$ is the set of executed test cases and $\{y,\ z\}$ is the set of candidates, then Jiang et al.'s approach [26] will treat y and z equally and randomly choose one of them to be the next test case; whereas the approach presented in the present paper will consider z to be farther apart from x and, therefore, select z to be the next test case. This treatment agrees with the intuition that z (rather than y) differs more from x because the execution profiles of x and z do not share any common value on any branch, but the execution profiles of x and y share common value "0" on three branches, namely on b_3, b_4 and b_5.

ART was developed as an enhancement to RT with an objective of reducing the number of test cases to reveal the first failure. A related technique, known as *adaptive testing*, was developed by Cai et al. [27]. Following the idea of adaptive control, adaptive testing adjusts the selections of test actions online to achieve an optimization objective, such as minimizing the total cost of revealing and removing multiple faults.

VII. Discussions and Conclusion

ART distance measures focused on input values are limited to programs that accept prescribed types of inputs. This paper proposed a general method to measure the distance between test cases based on coverage information. This kind of method enables ART to be applied to any programs without any limitation on input types. We then focused on a specific type of coverage information, the branch coverage, and conducted empirical evaluations. Experimental results with *replace* and *space* programs show that the proposed method improves the fault-detection capability of random testing significantly.

The proposed method can be directly used to prioritize regression test cases. Future research should study the usefulness of other types of coverage information for ART, including coverage of elements in function call graphs and in other types of graphs/models used in modelling systems, such as state diagrams and other UML diagrams. Furthermore, the coverage Manhattan distance studied in this paper only concerns coverage without counting frequency. Other measures involving frequency information should be investigated. For example, we can also use Equation (1) to define a *Frequency Manhattan Distance* by changing the meaning of x_i and y_i from "coverage" to "frequency".

Apart from regression testing, the proposed method can be applied to various test case generation tools. For example, some modern white-box testing tools such as CREST [2] use some random strategy to select the paths to be covered, and then use *symbolic execution* techniques to generate test cases to cover the selected paths. We expect that the proposed method can help improve the effectiveness of the path selection strategies in these tools. Indeed, the proposed method is not limited to white-box testing or node/edge coverage of graphs since coverage information can be derived from almost all kinds of models for systems and software.

A concern with ART is its time complexity in test case selection. Compared with test case generation, however, it is often more expensive or time consuming to run a test or to verify a test result. In these situations, it is highly desirable to have a strategy that can reduce the number of required test case executions, and ART helps to achieve this goal.

In this paper, we only studied one ART algorithm, namely FSCS-ART. There are other ART algorithms with lower time complexity. A future research topic is to investigate the feasibility of applying coverage-based distance measures to these ART algorithms.

VIII. Acknowledgments

The author would like to thank De Hao Huang for discussions on difference measures. This project is supported in part by a Small Grant of the University of Wollongong.

REFERENCES

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

[2] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society Press, 2008, pp. 443–446.

[3] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. John Wiley & Sons, 2002, pp. 970–978.

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

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

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[7] G. F. Renfer, "Automatic program testing," in *Proceedings of the 3rd Conference of the Computing and Data Processing Society of Canada*. University of Toronto Press, 1962.

[8] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

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

[10] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000, pp. 59–68.

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

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

[13] T. Y. Chen, T. H. Tse, and Y. T. Yu, "Proportional sampling strategy: A compendium and some insights," *Journal of Systems and Software*, vol. 58, pp. 65–81, 2001.

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

[15] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, "Adaptive random testing through iterative partitioning," in *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe 2006), Lecture Notes in Computer Science 4006*. Springer-Verlag, 2006, pp. 155–166.

[16] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008.

[17] T. Y. Chen and R. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 3, pp. 16:1–16:27, 2008.

[18] H. Jaygarl, C. K. Chang, and S. Kim, "Practical extensions of a randomized testing tool," in *Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society Press, 2009, pp. 148–153.

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

[20] R. G. Merkel, "Analysis and enhancements of adaptive random testing," Ph.D. dissertation, Swinburne University of Technology, Melbourne, Australia, 2005.

[21] F.-C. Kuo, "On adaptive random testing," Ph.D. dissertation, Swinburne University of Technology, Melbourne, Australia, 2006.

[22] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[23] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*. IEEE Computer Society Press, 1994, pp. 191–200.

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

[26] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society Press, November 2009, pp. 233–244.

[27] K.-Y. Cai, B. Gu, H. Hu, and Y.-C. Li, "Adaptive software testing with fixed-memory feedback," *Journal of Systems and Software*, vol. 80, pp. 1328–1348, 2007.