USING T-WAY INTERACTION TECHNIQUES FOR THE REDUCTION IN THE NUMBER OF TEST CASES

Ajiboye, Adeleke Raheem[1], Mejabi, Omenogo Veronica[2] & Salihu, Shakirat Aderonke[3]

[1,3]Department of Computer Science, Faculty of Communication & Information Sciences, University of Ilorin, Ilorin, Nigeria.

[2]Department of Information & Comm. Science, Faculty of Communication & Information Sciences, University of Ilorin, Ilorin, Nigeria.

[1] *ajibraheem@live.com* [2]*ovmejabi@gmail.com* [3] *shaksoft@yahoo.com*

ABSTRACT

A test case is a set of input data designed to discover a particular type of error or defect in the software system. In order to develop software that perform as expected, extensive testing should be carried out to ensure reliability. Ideally, software testers would want to test every possible permutation of the software, but in practice, due to the complexity of the software, exhaustive testing is usually not feasible. This paper presents the use of t-way interaction techniques with a view to reducing the number of test cases in the process of software testing. The software on which the approach is implemented consists of parameters that have the same number of values and their interaction is based on pairwise combination. The technique minimizes the number of test cases as it tests all pairs of variables. The resulting outputs show a significant reduction in the number of test cases from 8 to 6; this is a 25 % reduction. Thus, the overall time required to test the software is optimized. Also, the final reduced test cases are found to be free of redundancy and the technique used shows a high degree of parameter interaction.

Keywords: *Test cases, Test plan, Test suite, Software testing, T-way pairwise interaction.*

1. INTRODUCTION

Testing is one the several ways that can be used to measure and verify product quality. Software is a set of instructions used to acquire set of inputs and to manipulate these acquired inputs for the production of the desired output in terms of functions and performance as determined by the user (Agarwal et al., 2010). In recent times, everyone uses software in one way or another for convenience and better output. We use software in our homes, for business transactions, at the hospitals, and in a number of several consumer products such as cars, mobile phones water dispensers and washing machines. This is why it is said that, it is quite hard to imagine life without software (Zamli & Alkazemi, 2015). However, sometimes software may not work as expected as a result of errors or defects. Such errors for instance, may cause the software to fail unexpectedly. The cost of such sudden failure may be too much to bear and may possibly results to death. Proper testing of software is a promising way to avoid risk, especially risk that is related to human life.

Testing is a process rather than a single activity as it involves series of activities. In the process of testing software, the software to be tested is executed with a finite set of test cases, and the behaviour of the system for these test cases is evaluated. The purpose of such evaluation is to determine if the performance of the system meets the expectations. One of the benefits of ensuring testing testing of software is to give confidence to the developer team and the users against sudden failure. Although, testing do take place at each stage of software development, further test by independent testers at the end of the software development is crucial to achieving reliable software. The act of software testing helps to measure the quality of software especially as regards to the number of defects found, the tests run, and the system covered by the tests.

In software development, several tasks are performed at different stages and testing is vital to each stage. One of the most important things that should be considered in software testing is the design and creation of

effective test cases (Myers, 2004). But due to several parameters that one may need to test in a system, it becomes very important to reduce the number of test cases. In the course of doing this reduction, efforts are made to ensure redundancies are eliminated without sacrificing the reliability of the software being tested.

A test case is a description of conditions and expected results that are taken together in order to fully test a requirement (Fournier, 2009). Ideally, testers would prefer to have all aspects of the software to be covered during testing, but in most cases, especially when the software is complex, exhaustive testing though desirable, but is simply not feasible.

In view of the complexity of software to be tested, a meta-heuristic search technique for testing is a realistic approach, this is because, testing every part of the software is practically impossible (Afzal et al., 2008). The only obvious strategy is to achieve enough tests that can be a representative of the whole test.

In order to address this challenge, the use of sampling testing techniques such as boundary value analysis, equivalence partitioning, decision tables and random testing have yielded some acceptable results (Reid, 1997). Although, in terms of the likelihood of detecting the most errors in software, the use of random selection to generate test cases have narrow chance of achieving an optimal, or close to optimal subset. This is because the segment that is tested may not be a good representative of the whole partition.

The use of t-way interaction techniques used in this study gives much better variable interaction and minimizes the test cases. A number of t-way strategies exist and these strategies adopt either algebraic or computational approaches. The objective of this study is to demonstrate the use of t-way testing strategy with a view to minimizing the number of test cases that is required for the testing of software that has uniform values. Attempt to minimize test cases usually translate to a reduction in the overall cost of software testing.

The rest of the paper is structured as follows. In the next section, some of the related works reported in the literature is reviewed. This is followed by a brief discussion on the rationales for minimizing test cases, followed by a section that discusses software testing plans. Next to this section is the methodology used for the minimization of the test cases and in Section 6, the result generated is discussed, while the study is concluded in Section 7.

2. RELATED WORK

A number of test cases reduction strategies have been reported in the literature, some of them are reviewed here. Since it is generally known that exhaustive testing of software is impossible, most especially the complex ones. Researchers have therefore, focused on the possible ways to have the number of test cases minimized. In order to generate a test suite for software testing, study in (Bryce & Colbourn, 2007), combines a greedy algorithm with heuristic search. The study reported that, the suites generated dispense one test at a time and has a good coverage of variables.

In an attempt to minimize the number of test sets, the study in (Klaib et al., 2010) proposed a tree generation strategy for pairwise combinatorial software testing. The technique uses a cost calculation approach iteratively and all the leaf nodes were considered. The proposed approach was reported to have been used to generate the test suite until all the combinations were covered. A significant reduction in the number of test cases was also reported.

Also, in order to reduce the size of test suites generated for testing purposes, the study described in (Chen et al., 2010), applied particle swarm optimization technique which is a kind of meta-heuristic search technique to pairwise testing. The study further proposed two different algorithms for efficient generation of test cases.

Furthermore, the study reported in (Yan & Zhang, 2006), proposed along with other techniques, a backtracking algorithm and search heuristics. Both techniques were found useful in the generation of test suites for

combinatorial testing. The study reported that the proposed method was suitable and efficient in the reduction of the size of test cases.

Also, a related study by Blue et al. (2013) uses minimization technique to achieve a better test design. The study as reported in (Blue et al., 2013), specifically focuses on using the interaction-based test-suite minimization approach as a way of standardizing combinatorial test design.

A number of sampling methods for the minimization of test cases have also been reported in the literature. Typical examples of the techniques in this category include: equivalence partitioning, boundary value analysis and random testing. Reid (Reid, 1997), compares some of these methods and found boundary value analysis to be the most effective among the techniques that were investigated.

The study reported in (Tracey et al., 1998), relies on the use of sampling methods for the construction of an automated framework for test data generation. Equivalent Partitioning (EP) technique is an example of sampling method. EP technique reduces the number of test sets since the values in an equivalence partition are handled similarly. Testing only one part is assumed to have catered for the entire partition. One of the challenges with the use of sampling technique is the possibility of not achieving the optimum results. This is because the sample data used for testing may not be a good representative of the domain test data.

3. RATIONALES FOR MINIMIZING TEST CASES

A test case can be defined as a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works as expected (STF, 2016). Due to a combinatorial explosion in

software testing, it is necessary to carefully minimize test cases to a manageable size that can still give optimum results. For instance, considering software of 15 parameters, 14 of these parameters have 2 values each and the 15^{th} parameter has 65 values; this may be a set of different colours. This would give a combinatorial explosion that can take several years to test: $2^{14} \times 65 = 16{,}384 \times 65 = 1{,}064{,}960$ combinations.

Since it is practically impossible to test large combinations like this exhaustively, then a minimized test set is inevitable. Some reasons why software may not work as expected include the environmental conditions like the presence of radiation, magnetism, electronic fields or pollution (Morgan et al.). Further clarifications made in (Morgan et al., 2010) shows that these factors are capable of changing the way hardware and firmware operate. Such sudden change can lead to system failure. In order to avoid system failure, therefore, we should critically look for errors and faults in a system with a view to rectifying them.

One of the implications of software that is not working correctly is the possible potential of causing harm to people. It can make a company to record high loss and can lead to pollution of the environment (Morgan et al., 2010). For instance, incorrect billing can lead to huge losses to a company, while the release of radiation or poisonous chemicals to the environment can cause serious disaster.

The procedure involved in making high-quality products is very tasking. Although, certain costs may be obvious, like sales, marketing, and employee salaries; other costs, especially indirect ones such as employee training, testing, fixing errors, retesting, and reporting on information, may be less obvious and are often neglected by organizations; and of these, testing appears to be the most bemoaned (Delesie, 2012).

There are several procedures that must be followed in the course of designing test cases. Even after selecting the set of test cases necessary to cover the identified risks, there are still other things that should also be considered necessary. Test cases may need to be changed, adapted, or

added and in addition, careful attention must be given to ensure that test data can be reused (Rosink, 2012). In this phase, effort is expended in setting up the test data for test cases.

The writing of test cases should encompass the description of the main functionality that needs to be tested and the readiness to assure that the test can be conducted. What make up the test cases basically includes the set of input values, execution preconditions, expected results, and execution post conditions, developed for a particular objective or test condition; for instance, to exercise a particular program path or to verify compliance with a specific requirement (Homès, 2012).

Test planning that focuses on creation and design of test cases is perhaps the most pivotal aspect of software testing (Zamli & Alkazemi, 2015). The reason for this is that, a test design that is poorly planned is capable of making the bug detection in the system difficult. Test-case design is so important because it is not possible to test all; and it follows that, a test of any program must be necessarily incomplete (Myers, 2004).

4. SOFTWARE TESTING PLANS

Planning involves what is going to be tested, and how this will be achieved. A test plan is a document consisting of different test cases designed for different testing objects and different testing attributes (Agarwal et al., 2010). It is where a map is drawn; how activities will be done; and who will do them. Test planning is also where the test completion criteria is defined. Completion criteria are how we know when testing is finished. The Plan describes the strategy for integration, puts the test in logical and sequential order per the strategy chosen, top-down or bottom-up. It is a matrix of test and test cases listed in order of its execution. The test to be carried out should be planned long before testing begins. In most cases, test plan begins shortly after the requirements model is completed.

The conceptual test cases are expected to be defined when the product features and requirements
are specified. They should be included in the plan, and the time for testing and debugging each feature should be added to the project estimates (Huizinga & Kolawa, 2007). Also, part of a good plan is to begin detailed test cases as soon as the design model is concluded. A test plan basically states the items to be tested, at what level they will be tested, the sequence they are to be tested, and how the test strategy will be applied to the testing of each item and the test environment. Table 1, illustrates the matrix of test and test cases within the test adapted from Agarwal et al. (2010).

Test coverage measures in some specific way the amount of testing performed by a set of tests.
It is an important task that must be adequately planned in the course of testing. Test coverage is all about having enough tests to sufficiently demonstrate that the system meets specifications as perceived by the stakeholders (Fournier, 2009). It provides a quantitative assessment of the extent and quality of testing. In other words, it answers the question 'how much testing have you done?' in such a way that it is not open to interpretation. Test coverage can be measured based on a number of different structural elements in a system or component. Coverage can be measured at component-testing level, integration-testing level or at system- or acceptance-testing levels.

Table 1: Test Plan Format

Test									
Test ID		Test	1	2	3	...	N	Planned Date	
ID	Tester	Name						Completed	Successful

5. MATERIAL AND METHODS

T-way pairwise interaction technique is used in this study for the reduction in the number of test cases. There are a number of methods that can be used under t-way combinatorial techniques. The number of values is one of the determinants of the approach that is most suitable. Since the software under test represented in Figure 1, has uniform values, t- way pairwise interaction appears to be a suitable technique. The number of interactions, t, is taken as 2. Figure 1, is an excerpts from a software system and the users are expected to choose a value from each parameter to place an order. The single interface under test has 3 parameters each with 2 values. That is, $X = x_1, x_2$; $Y = y_1, y_2$; $Z = z_1, z_2$. The values: x_1, x_2; y_1, y_2 and z_1, z_2 are referred to as base values and assigned to the parameters as Tablet = {Ibuprofen (x_1), Diclofenac (x_2)};
Cream = {Neurogesic (y_1), Volini (y_2)};
Injection = {Penthazocin (z_1), Tramadol (z_2)}.

A test suite is constructed with these parameters as represented in Table 2. The mathematical notation of the technique used conforms to the expression in (1):

$$CA\ (N,t,V^p) \qquad (1)$$

where CA is the Covering Array, N is the size of test suite, *t* is the number of interactions. In this instance, *t* = 2.
V is the number of values.
p is the number of parameters.

Therefore, using covering array and taking the interaction, t = 2, the number of test cases that can be generated for exhaustive testing is V^p. Since the number of values for each parameter is 2, the exhaustive testing for the interface represented in Figure 1, becomes 2^3, which equals 8 test cases. The exhaustive combination is illustrated in Table 2.

It should be noted that the possible techniques that can be used under t-way interaction is not limited to uniform strength interaction used in this study. Other known interaction strategies are: cumulative strength, variable strength and input-output based relation. In the course of choosing any of these techniques, the parameters and the distribution of values is usually taken into consideration. The tables 2, 3, 4 and 5 are the general template for an interface of 3 parameters, each with a pair of uniform values. Apart from using x, y, and z, any other characters can also be used to denote the values; digits can also be used instead as far as it brings about clarity.

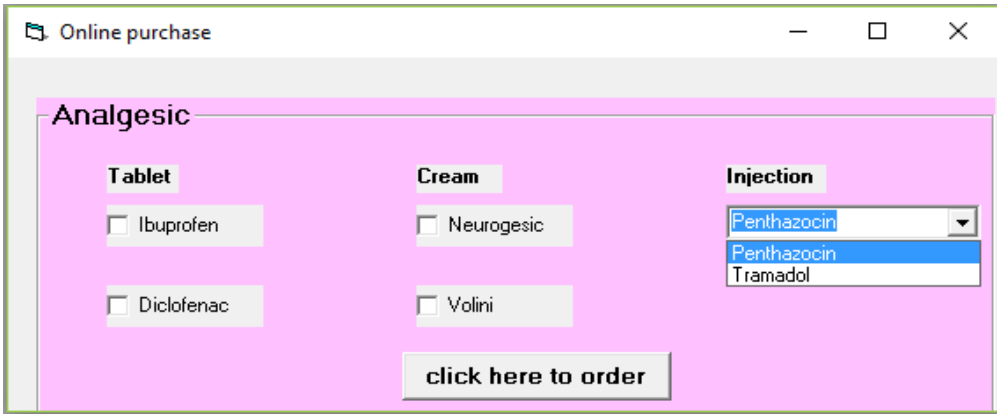

Figure 1. Interface showing parameters of uniform values

Table 2. Exhaustive 2-way combination

	Input		
	X	Y	Z
Base Values	x_1	y_1	z_1
	x_2	y_2	z_2
Exhaustive Combinations	x_1	y_1	z_1
	x_1	y_1	z_2
	x_1	y_2	z_1
	x_1	y_2	z_2
	x_2	y_1	z_1
	x_2	y_1	z_2
	x_2	y_2	z_1
	x_2	y_2	z_2

Considering the parameters in Figure 1: XYZ and the values of each parameter X = {x_1, x_2}; Y = {y_1,y_2} and Z = {z_1,z_2}, the equivalent test set for exhaustive testing using pairwise combinatorial is as follows:

1: { Ibuprofen, Neurogesic, Penthazocin}
2: { Ibuprofen, Neurogesic, Tramadol}
3: { Ibuprofen, Volini, Penthazocin}
4: { Ibuprofen, Volini, Tramadol}
5: { Diclofenac, Neurogesic, Penthazocin}
6: { Diclofenac, Neurogesic, Tramadol}
7: { Diclofenac, Volini, Penthazocin}
8: { Diclofenac, Volini, Tramadol}

For any interface that has three parameters, for instance, the XYZ as illustrated in Figure 1, the possible permutations are: XY, XZ and YZ. Each of these combinations forms the set of test cases. What is unique about 2-way, otherwise known as pairwise testing technique is that, the pairwise technique minimizes the number of test cases by testing all pairs of variables. Thus, in the present study, the combination of each pair of variables gives the values shown in Tables 3, 4 and 5.

The number of occurrences in each pair of combination is determined. The focus is to ensure that no variable is uncovered, while those that appear more than once are removed to avoid redundancy as illustrated in Tables 3 and 4. The leftover, that is, those that appear only once form the final test suite shown in the final result section (see Table 6). Although, there are several software testing tools that can be used to generate test suite, however, this procedure is implemented using Matlab codes. The implementation involves ensuring that all the redundancies in the test cases are eliminated. For instance, in Table 3, row XY is splitted into 4, the same pattern is followed in Tables 4 and 5, where rows XZ and YZ are splitted into 4 as shown. Looking at all the tables as one, the similar test cases are cancelled out to give the unique test cases represented in Table 6.

Table 3. The combination of parameters X and Y

	Input		
	X	Y	Z
Base Values	x_1	y_1	z_1
	x_2	y_2	z_2
XY	x_1	y_1	z_1
	x_1	y_2	z_1
	x_2	y_1	z_2
	x_2	y_2	z_2

Table 4. The combination of parameters X and Z

	Input		
	X	Y	Z
Base Values	x_1	y_1	z_1
	x_2	y_2	z_2
XZ	x_1	y_1	z_1
	x_1	y_1	z_2
	x_2	y_2	z_1
	x_2	y_2	z_2

Table 5. The combination of parameters Y and Z

	Input		
	X	Y	Z
Base Values	x_1	y_1	z_1
	x_2	y_2	z_2
YZ	x_1	y_1	z_1
	x_2	y_1	z_2
	x_1	y_2	z_1
	x_2	y_2	z_2

6. RESULTS AND DISCUSSION

This section discusses the test suite generated from the pairwise interaction of parameters under test. From the initial exhaustive testing, the study further reduces the number of test cases. The final unique test cases that is free of redundants. The result shows some significant reduction in the number of test cases from an initial test cases of size 8 as earlier shown in Table 2, to 6 test cases as represented in Table 6. This is equivalent to a 25% minimization of the number of test cases.

Those test cases that occurred twice, which could have resulted to duplications have been removed from the permutation shown in Tables 4 and 5. Only the unique test cases are left and harnessed together to form Table 6. The selection of the parameter values shown earlier in Figure 1, can then be made based on these final test cases without ignoring or making repetition in the test design.

The selection of test cases in the software excerpts represented in Figure 1, gives the exhaustive combinations of 3 parameters, each with 2 uniform values. The interaction of the parameters based on the possible permutations and subsequent elimination of redundancies results to unique test sets. The initial test suite is then minimized to the following test cases as listed in numbers 1 - 6:

1: { Ibuprofen, Neurogesic, Penthazocin}
2: { Ibuprofen, Neurogesic, Tramadol}
3: { Ibuprofen, Volini, Penthazocin}
4: { Diclofenac, Neurogesic, Penthazocin}
5: { Diclofenac, Neurogesic, Tramadol}
6: { Diclofenac, Volini, Tramadol}

Table 6. The final minimized test cases generated

	Input		
	X	Y	Z
Base Values	x_1	y_1	z_1
	x_2	y_2	z_2
Pairwise Combinatorial	x_1	y_1	z_1
	x_1	y_1	z_2
	x_1	y_2	z_1
	x_2	y_1	z_1
	x_2	y_1	z_2
	x_2	y_2	z_2

7. CONCLUSION

This paper has shown how the t-way pairwise interaction techniques can be used to optimize test cases. There are so many approaches of minimizing the number of test cases, but the choice of a particular technique depends on a number of factors, such as the parameters and number of values involved. This study focuses on reducing the number of input data required to carry out testing of a section of software (unit testing). This is achieved in this study without sacrificing the level of accuracy of the software under test. Unit testing involves testing of individual components to ensure that they operate correctly. When a test is carried out on software, such effort can only show that one or more defects exist. Testing is not capable of showing that the software under test is error free. The essence of software testing is, therefore, to greatly reduce risk or sudden failure of the software.

Due to the challenges one is bound to face when the software is to be tested exhaustively, the most feasible approach is to minimize the test sets without sacrificing the reliability of the software. Testing unveils some inconsistencies and bugs that may find its way into the software. In reality, software that consists of large parameters of non-uniform values can take some decades to test, hence, the need to have the test sets minimized.

Although the interface under test represented earlier in Figure 1, has a simple logical structure, it is for illustrative purpose, as it consists of 3 parameters of 2 uniform values only; however, a larger number of parameters of uniform values follow the same procedures shown in this paper. The results from this study show that, the number of test sets generated for the exhaustive combinations of the parameters have been reduced by 25%. Apart from the fact that the approach eliminates some duplication that is capable of swelling up the number of test cases, thereby saving testing time, the technique is found to be efficient in the minimization of test sets.

ACKNOWLEDGMENT

The Authors would like to thank the anonymous reviewers for their useful comments and suggestions on this paper.

REFERENCES

Afzal, W., Torkar, R., & Feldt, R. (2008). A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*.

Agarwal, B. B., Tayal, S. P., & Gupta, M. (2010). *Software Engineering & Testing*: Jones and Bartlett Publishers.

Blue, D., Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2013). *Interaction-based test-suite minimization.* Paper presented at the International Conference on Software Engineering.

Bryce, R. C., & Colbourn, C. J. (2007). *One-test-at-a-time heuristic search for interaction test suites.* Paper presented at the 9th annual conference on Genetic and evolutionary computation.

Chen, X., Gu, Q., Qi, J., & Chen, D. (2010). *Applying particle swarm optimization to pairwise testing.* Paper presented at the 34th IEEE Annual Computer Software and Applications Conference.

Delesie, S. (2012). *How to Reduce the Cost of Software Quality*: CRC Press, Taylor & Francis Group

Fournier, G. (2009). *Essential Software Testing A Use-Case Approach*: CRC Tailor & Francis Group.

Homès, B. (2012). *Fundamentals of Software Testing*. UK: ISTE Ltd

Huizinga, D., & Kolawa, A. (2007). *AUTOMATED DEFECT PREVENTION: Best Practices in Software Management*. New Jersey: John Wiley & Sons, Inc.

Klaib, M. F., Muthuraman, S., Ahmad, N., & Sidek, R. (2010). Tree based test case generation and cost calculation strategy for uniform parametric pairwise testing. *Journal of Computer Science, 6*(5), 542.

Morgan, P., Samaroo, A., Thompson, G., & Willams, P. (2010). *SOFTWARE TESTING An ISTQB-ISEB Foundation Guide* (B. Hambling Ed. Second ed.): British Informatics Society Limited.

Myers, G. J. (2004). *The Art of Software Testing* (Second ed.): John Wiley & Sons.

Reid, S. C. (1997). *An empirical analysis of equivalence partitioning, boundary value analysis and random testing.* Paper presented at the Fourth International Software Metrics Symposium.

Rosink, J. (2012). *How to Reduce the Cost of Software Testing*: CRC Press, Taylor & Francis Group.

STF. (2016). Software Testing Fundamentals (STF). 2016, retreived from http://softwaretestingfundamentals.com/test-case/ on Jan 22, 2017.

Tracey, N., Clark, J., Mander, K., & McDermid, J. (1998). *An automated framework for structural test-data generation.* Paper presented at the 13th IEEE International Conference on Automated Software Engineering, .

Yan, J., & Zhang, J. (2006). *Backtracking algorithms and search heuristics to generate test suites for combinatorial testing.* Paper presented at the 30th Annual International Conference on Computer Software and Applications

Zamli, K. Z., & Alkazemi, B. Y. (2015). *Combinatorial T-way Testing*. Malaysia: UMP.