2011 Fifth International Conference on Secure Software Integration and Reliability Improvement

On Testing Effectiveness of Metamorphic Relations: A Case Study

Mahmuda Asrafi, Huai Liu*, Fei-Ching Kuo
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn 3122 VIC, Australia
e-mail: {masrafi, hliu, dkuo}@swin.edu.au

***Abstract*—One fundamental challenge for software testing is the oracle problem, which means that either there does not exist a mechanism (called oracle) to verify the test output given any possible program input, or it is very expensive, if not impossible, to apply the oracle. Metamorphic testing is an innovative approach to oracle problem. In metamorphic testing, metamorphic relations are derived from the innate characteristics of the software under test. These relations can help to generate test data and verify the correctness of the test result without the need of oracle. The effectiveness of metamorphic relations can play a significant role in the testing process. It has been argued that the metamorphic relations that cause different software execution behaviors should have high fault detection ability. In this paper, we conduct a case study to analyze the relationship between the execution behavior and the fault-detection effectiveness of metamorphic relations. Some code coverage criteria are used to reflect the execution behavior. It is shown that there is a certain degree of correlation between the code coverage achieved by a metamorphic relation and its fault-detection effectiveness.**

Keywords- software testing, metamorphic testing, metamorphic relation, fault-detection effectiveness, code coverage.

I. INTRODUCTION

Software testing is a very crucial approach for assuring the quality of the software applications. Although software testing cannot guarantee the absence of faults, it is the major approach of revealing software faults. After the testing process, other techniques such as debugging can be applied to fix the faults and thus to improve the software quality. Many testing methods have been proposed to select some program inputs as test cases such that faults in the program can be effectively detected. After executing the test cases, the test outputs are checked against a test oracle to verify the functionality of the system. An oracle is a mechanism for determining whether the program has passed or failed a test. A complete oracle should be accomplished of an "originator", a "comparator" and an "evaluator" [17]. Originator offers the expected outcome for each test case. Comparator checks the test output against the expected outcome. Finally evaluator verifies whether the software under test has passed or failed the testing.

Without the presence of an oracle it is very difficult to verify the correctness of test outputs and thus the effectiveness of software testing is greatly hindered. This dilemma is known as "oracle problem" in software engineering. It is one of the most difficult tasks in software testing [23]. An effort to resolve this problem is to use a pseudo-oracle [13], where several implementations of an algorithm practice an input and the outcomes are compared to decide whether there are faults in some of the implementations. However, this procedure is not useful in many practical situations where various implementations may not exist or may be created by same group of developers who tend to make same fault. Several other techniques such as gold standard oracle [1], reference model [5, 6], assertion checking [2, 26], and metamorphic testing [7] have been proposed to alleviate the oracle problem.

Metamorphic testing [7] employs properties of the target function of the software under test. It discovers some properties from the specification or algorithm of the software under test. Based on these properties some relations are derived known as metamorphic relations (MRs). Some source test cases from traditional test case generation methods are used in metamorphic testing. MRs are used to generate follow-up test cases based on the source test cases. After executing source and follow-up test cases metamorphic testing verifies the outputs of test cases based on MRs. A great amount of MRs can be identified from the algorithm or specification of the software under test. Different MRs have different effectiveness for the detection of various faults. It is important to use MRs with high fault-detection effectiveness to save the time and resources.

It has been suggested [10] that MRs that can cause the program under test to exhibit diverse execution behaviors should have high fault-detection effectiveness. In this paper we conduct a case study to investigate to what extent the execution behaviors caused by an MR is correlated with its effectiveness. It is expected that the study will result in some rules to judge the effectiveness of MRs.

This paper is organized as follows. Section II presents the basic information of metamorphic testing. Section III introduces some previous studies related to metamorphic testing and the selection of good MRs. Section IV reports our case study and discuss the experimental results. Section V

* Corresponding author.

978-0-7695-4453-3/11 $25.00 © 2011 IEEE
DOI 10.1109/SSIRI.2011.21

discusses the threats to validity. Section VI concludes the paper.

II. METAMORPHIC TESTING

Metamorphic testing (MT) is an innovative approach for alleviating the oracle problem. It aims to conduct the testing on the basis of some domain knowledge acquired from the algorithm or specification of the software under test. A metamorphic relation (MR) is an expected relation of the software under test which should be valid over a set of distinct input data and their corresponding output values for multiple executions. MT checks the validity of MRs by multiple executions of the target program. MT is conducted as follows: (1) find out specific properties of the SUT to construct MRs, (2) generate source test case by some traditional testing techniques (such as random testing, fault-based testing, etc), (3) generate follow-up test cases based on source test cases according to the MRs, (4) execute the test cases, and (5) verify the outputs of the test cases against MRs. If the outputs of the source and follow-up test cases violate their corresponding MR, then a fault is detected.

A simple example to elaborate the MT technique is a sorting program, which sorts a set of integers in the ascending order. Suppose S is a set of elements to be sorted. If the set S is rearranged in reverse order the output of the sorting program will still remain same. This MR can be denoted by Sort(S) = Sort (reverse(S)). Suppose S = {35, 15, 32, 25}, Sort(S) will yield {15, 25, 32, 35}.We reverse the set S to generate the follow-up test case reverse(S) = {25, 32, 15, 35}. If Sort (reverse(S)) ≠ {15, 25, 32, 35}, we can say a fault is detected.

III. RELATED WORK

Since the proposal of MT, it has been applied to detect faults in various areas. MT was first used for testing scientific programs [7]. Distributional properties have been used in [24] as MRs to test image processing and analysis applications. MT is used to find errors in a program solving elliptic partial differential equations with dirichlet boundary conditions [8]. Isotropic properties of contexts were used as MRs for testing context-sensitive applications [28] and this work was enhanced further by using checkpoints [4]. Real life bugs were detected by MT in some bioinformatics programs [9].

Some testing methodologies were proposed based on MT. A combination of MT and symbolic execution namely semi-proving [11], uses symbolic inputs to indicate whether a program satisfies a MR (at least for an execution path). Some approaches have been proposed to automate MT [14, 27]. MT was also integrated with fault-based testing [12].

The major task in MT is to identify proper MRs. Mayer and Guderlei [25] conducted case studies to check the effectiveness of different MRs. They found that MRs with rich semantic properties are typically strong and proposed that testers should not select MRs that are too close to the implemented algorithm. Chen et al. [10] conducted case studies for selection of good MRs where MT is applied on implementations of shortest path and critical path algorithms. It was suggested that the theoretical properties are not sufficient to distinguish good MRs, while MRs that can make the executions of the software under test more different are good MRs [10]. It was proposed to understand the algorithm of the software under test before selecting MRs.

Having said that, no work has been conducted to systematically evaluate the relationship between the execution behaviors and the fault-detection effectiveness of MRs. In this paper, we conduct a case study to examine to what extent the execution behaviors caused by an MR are correlated to its fault-detection effectiveness.

IV. CASE STUDY

We have conducted some experiments to investigate the relationship between the resultant execution behaviors and the effectiveness of MRs. In our study, execution behaviors are measured against the code coverage achieved by the source and corresponding follow-up test cases. The rationale behind this measurement is that the higher the code coverage the more diverse the execution behaviors of the test set.

A. Subject programs

Table I: Subject Programs

Program	Language	LOC
TCAS	C	173
KNASPSACK	Java	780

In this study, we have selected two programs as the subject. One program is TCAS [15] which is written in C language. TCAS is an implementation of onboard aircraft conflict detection and resolution system. It accepts twelve input parameters, judges whether there will be a conflict between the current aircraft and the intruder aircraft based on the inputs, and finally outputs which kind of manoeuvre the current aircraft should take. TCAS has three types of outputs: 0 represents UNRESOLVED that indicates no manoeuvre, while 1 and 2 represent UPWARD or DOWNWARD manoeuvres, respectively.

The other subject program is KNAPSACK [19], which is written in Java language. The KNAPSACK program accepts three sets of integers. Two n-tuple sets $P = \{p_1, p_2, \ldots, p_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ represent the profits and the weights of n items, respectively; while another m-tuple set $C = \{c_1, c_2, \ldots, c_m\}$ contains the capacities of m knapsacks. The outputs of KNAPSACK are one n-tuple set $Y = \{y_1, y_2, \ldots, y_n\}$ and one positive integer TP. $y_i = j$ (where $i = 1, 2, \ldots, n$ and $j = 0, 1, \ldots, m$) represents that the i^{th} item should be put into the j^{th} knapsack. If $y_i = 0$, it means that the i^{th} item will not be selected into any knapsack. TP represents the total profit of the picked items. The KNAPSACK program attempts to calculate the optimal solution and thus to maximize the total profit.

B. Metamorphic Relations (MRs)

We have identified fourteen MRs for TCAS and ten MRs for KNAPSACK. During the identification of MRs, we have considered all the input parameters and all the functionalities of the subject programs. The identified MRs have covered most portions of the subject programs. Different MRs reflect different aspects of the subject programs, and thus have diversified characteristics.

Details of all the identified MRs can be found in Appendix. The following give two examples of the MRs.

- MR1-TCAS: Given that the intruder aircraft does not have the TCAS system, if we change the intention of the intruder aircraft, the outputs of the source and follow-up test cases should be identical.
- MR1-KNAPSACK: Given the source test case $T = \{P, W, C\}$, its output is $O = \{Y, TP\}$. Swap the k^{th} and the l^{th} items, where $1 \le k < l \le n$, and $p_k \ne p_l$ or $w_k \ne w_l$. We can get the follow-up test case $T' = \{P', W', C\}$, where $P' = \{p_1, p_2, \ldots, p_l, \ldots, p_k, \ldots, p_n\}$ and $W' = \{w_1, w_2, \ldots, w_l, \ldots, w_k, \ldots, w_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = \{y_1, y_2, \ldots, y_l, \ldots, y_k, \ldots, y_n\}$ and $TP' = TP$.

C. Fault detection effectiveness of MRs

Mutation analysis has been applied in this study to evaluate the testing effectiveness. Mutation analysis is a method where some faults are injected into the source code of the original program to generate some faulty versions, which are known as mutants. A set of test cases are normally executed on the original and its mutant programs. The output of the original program is compared against a mutant to detect any dissimilarity between the outputs for the same test case. If dissimilarity is found, then the mutant is said to be killed, and thus fault to be detected by the test case. In other words, the original program under test acts as the oracle to verify the correctness of the mutants in traditional mutation analysis. However, in our study, we will investigate the effectiveness of MRs without the need of oracle. We are using the MT technique to test the mutants generated from the subject programs.

We have generated 422 mutants for TCAS using automated mutant generator tool Milu [16]. We have tested them using all fourteen MRs identified in the previous section. For each MR, we used random testing technique to construct 10,000 source test cases, and then generated 10,000 corresponding follow-up tests cases according to each MR. The effectiveness of MRs is measured against the number of detected faults. After the execution of all test cases of all MRs, the numbers of faults detected by each MR are plotted in Fig.1. From Fig.1 it can be observed that among all MRs, MR4 detects the highest number of faults while MR2 detects only one fault. In addition, MR5, MR6, and MR7 also detect a great amount of faults comparing to other MRs for this program.

A total of 100 mutants are generated by automated mutant generator tool muJava [22] for KNAPSACK. We test them using all ten MRs identified for this program. The test case generation process is similar to that for TCAS.

After the execution of all test cases of all MRs the numbers of faults detected by each MR are plotted in Fig.2. From Fig.2, we can observe that except MR1 all the MRs can detect all the faults.

D. Coverage achieved by MRs

Code coverage is used to measure the degree to which the source code of a program has been executed. In this study, we are considering the coverage percentages achieved by the test cases of a MR as the representative of its carried out execution behaviors.

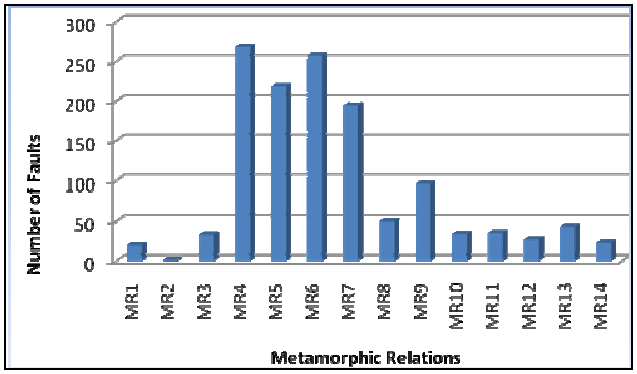

Fig.1: Number of Faults Detected by Each MR in TCAS

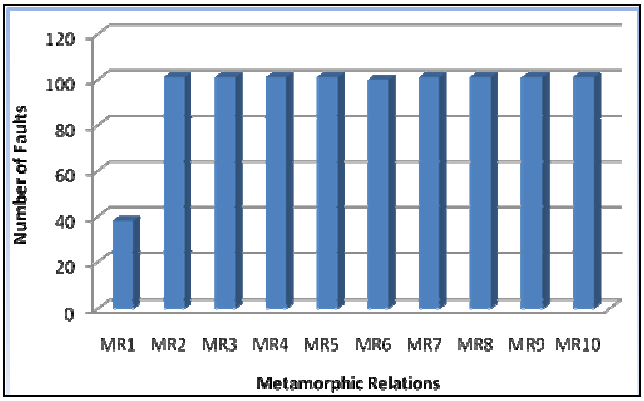

Fig.2: Number of Faults Detecetd by Each MR in KNAPSACK

Coverage data have been collected for both subject programs using automated coverage data collection tool. We collected the data for line and branch coverage. Line coverage is used to determine which lines/statements of the code are being covered throughout the execution of test cases. Branch coverage calculates the percentage of the branches in the code that are covered by test cases.

For TCAS, linux tools, namely gcov[21] and lcov[20], are used to collect coverage achieved by all MRs. gcov is a test coverage program which can be used as a profiling tool in connection with gcc to test code coverage in programs written in C. lcov is a graphical interface for gcov. It collects gcov data for multiple source files.

For KNAPSACK, a standalone application CodeCover [3] is used to collect line and branch coverage. CodeCover is a free white-box testing tool that measures statement, branch, loop, MC/DC operator, and sync- coverage.

All 10,000 source test cases and corresponding 10,000 follow up test cases used in the previous section for each metamorphic relation are executed on the subject program. The accumulative coverage percentages achieved by these 20,000 test cases are considered as the coverage achieved by that particular MR for the program. The same procedure has been applied on the mutant programs to collect the accumulative coverage data for each MR. Thus we have a set of coverage percentages achieved by each MR from the original and mutant programs. For example, KNAPSACK has 100 mutants, so each MR of this program has a set of 101 coverage percentages. The mode and average value of coverage set for each MR on all the programs are calculated. Here mode value refers to the coverage percentage that occurs most often in the set of all coverage values achieved by the particular MR. On the other hand, average value refers to the arithmetic mean of the coverage percentages achieved by a particular MR. These values are plotted in Figs. 3 to 6. Fig.3 and Fig.4 display the line coverage and branch coverage values achieved by fourteen MRs for TCAS, respectively. Fig.5 and Fig.6 represent the line and branch coverage data for KNAPSACK, respectively.

Based on these Figures, we made the following observations on the two subject programs:

1) *For program TCAS:*

a) Line Coverage: From Fig.3, we can observe that MR6 and MR7 have low line coverage, MR1 to MR5 have same percentage of line coverage. MR9, MR11 and MR13 have the highest line coverage among all the MRs.

b) Branch Coverage: From Fig.4, we can see that MR 4 to MR7 have the lowest branch coverage. The other MRs however maintain similar amount of high branch coverage. MR13 still maintains the highest branch coverage as it does in line coverage.

c) MODE vs AVE: Both in line and branch coverage we can see the MODE and AVE value of the coverage percentage achieved by the MRs is quite close to each other.

2) *For program KNAPSACK:*

a) Line Coverage: From Fig.5, it can be observed that MR9 has the highest line coverage that is 87.5%. On the other hand six MRs (that is, MR1 to MR4, MR6, and MR7), which are in a low stand in the figure in comparison to other MRs of this program, are bearing line coverage near 85%.

b) Branch Coverage: Based on Fig.6, we can observe MR7, MR9 and MR10 show higher branch coverage than other MRs. MR1, MR2, MR4 and MR 6 achieve lower branch coverage in comparison to other MRs. In contrast to line coverage, MR3 achieves high branch coverage here.

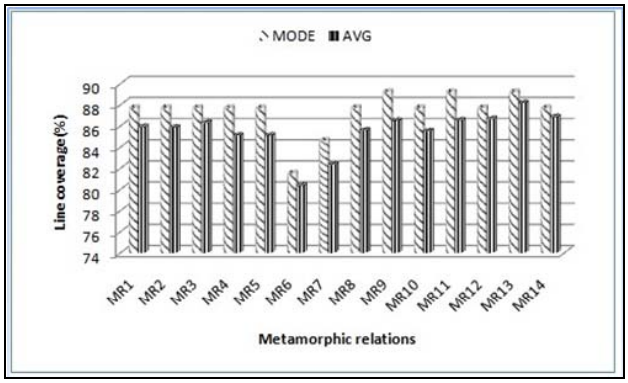

Fig.3: Line Coverage for TCAS

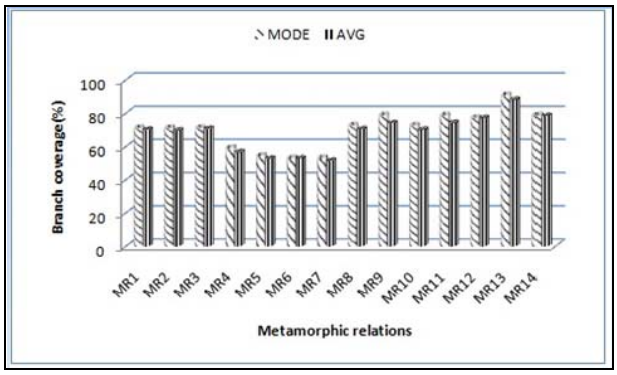

Fig.4: Branch Coverage for TCAS

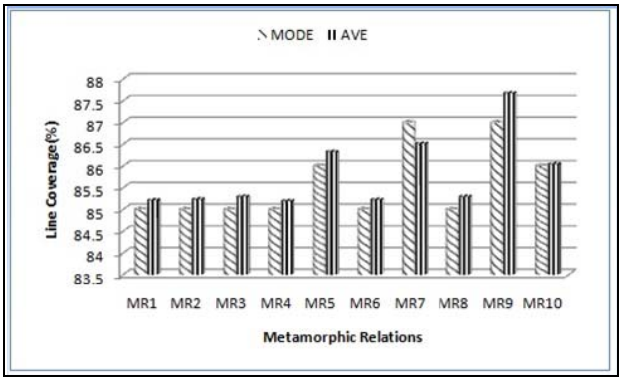

Fig.5: Line Coverage for KNAPSACK

c) MODE vs AVE: For line coverage we can see that MR7 has a higher mode value than its average value. In contrary MR9 has a higher average value than its corresponding mode value. From Fig.6 it is also visible that in MR7, MR9 and MR10 the average branch coverage value is a little bit higher than their corresponding mode value.

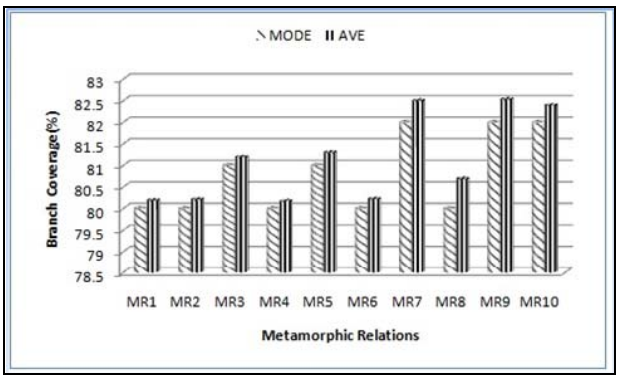

Fig.6: Branch Coverage for KNAPSACK

E. Discussion:Relationship between Code Coverage and Fault-detection effectiveness

Our identified MRs are associated with a large number of different coverage percentages as well as various numbers of detected faults. It is difficult to analyze the relationship between code coverage and fault-detection effectiveness directly on these raw data. In the following, we conduct some statistical analyses.

We have divided the whole range of coverage percentages (line and branch) in several clusters using standard histogram function [18]. The histogram function distributes the elements of the coverage set into equally spaced clusters and returns the number of elements in each cluster. The total number of MRs and the total number of fault detecting MRs in the particular range are calculated. Here we are considering each MR with its achieved coverage percentages on different mutant programs. For example, KNAPSACK has (10*101) set of coverage percentages achieved by its ten MRs on 100 mutants as well as the original KNAPSACK program.

We first define "MR Probability" (MRP) as the probability of a MR to be located in one particular range, as follows:

$$MRP = \frac{No.\ of\ MRs\ in\ the\ range}{Total\ No.\ of\ MRs\ \ for\ the\ program}$$

Second, we define "Fault Detection Probability within a Range" (FDPR), as the probability of a MR in one particular range to be able to detect faults, as follows:

$$FDPR = \frac{No.of\ Fault\ Detecting\ MRs\ in\ \ the\ range}{Total\ \ No.\ \ of\ MRs\ in\ the\ range}$$

Finally, we define "Fault Detection Probability" (FDP) as the probability of all MRs in one particular range to be able to detect faults, as follows:

$$FDP = MRP * FDPR$$

Figs. 7 to 10 illustrate the relationship between the coverage percentages and FDPs. In these figures, the x-axis represents the centre points of each cluster of coverage percentages, while the y-axis denotes the values of FDPs. For ease of illustration, we also drew an exponential trend line for the points in each figure.

To analyze to what extent these data are correlated with each other, we have calculated Pearson correlation coefficient. Pearson's correlation coefficient, which is based on the method of covariance, is a well-known method of measuring the correlation [18]. Pearson's correlation coefficient gives information about the degree of correlation as well as the direction of the correlation (as shown by the curves in Figs. 7 to 10). If Pearson's correlation coefficient value is near ± 1, then it said to be a perfect correlation. If Pearson's correlation coefficient value lies between ± 0.75 and ± 1, then it is said to be a high degree of correlation. The coefficient values for the points in Figs 7 to 10 are summarized in the Table II.

Table II: Pearson Correlation Co-efficient Values between Coverage Percentage and FDPs

Program	Line coverage Vs FDP coefficient	Branch coverage Vs FDP coefficient
TCAS	0.69	0.56
KNAPSACK	0.98	0.92

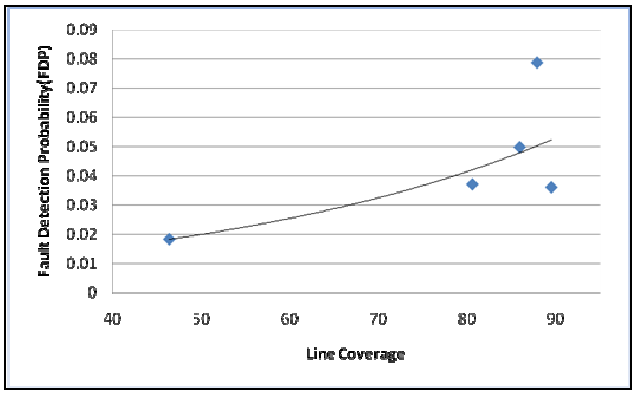

Fig.7: Fault Detection Probability of the MRs with Line Coverage for TCAS

Generally speaking, in both type of coverage criteria (line and branch) we can observe that FDP normally increases with the increase in coverage value. However, under some situations, the rising trend falls in some points at the higher coverage percentages. This phenomenon implies that some MRs with high coverage do not have high fault-detection effectiveness. Detailed of these scenarios along with the figures of each subject program are explained as follows.

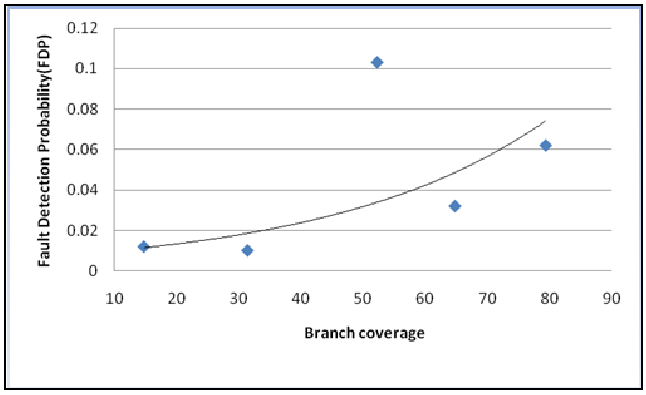

Fig.8: Fault Detection Probability of the MRs with Branch Coverage for TCAS

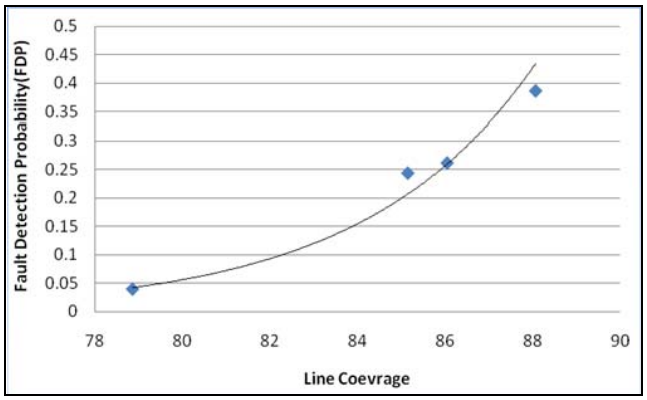

Fig.9: Fault Detection Probability of the MRs with Line Coverage for KNAPSACK

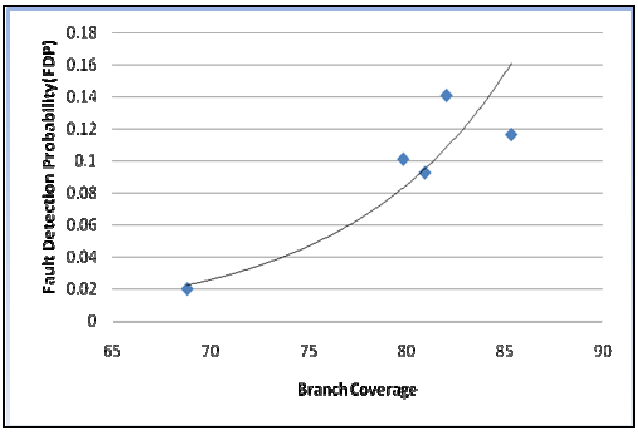

Fig.10: Fault Detection Probability of the MRs with Branch Coverage for KNAPSACK

For TCAS, rising trend is not so constant. In Fig.7, we found that the rightmost two points are inconsistent with the rising trend. Further inspection shows that the numbers of samples in the ranges represented by these two points differ 1.5 times. This may cause to lower down the value of FDP for the rightmost point with line coverage 89.52% from the other point with line coverage 87.90%. Again in Fig.8, for branch coverage large fluctuation is visible for third and fourth points from left. Here we found that the number of sample for range represented by the third point (with branch coverage 52.25%) is 2.3 times more than that for the fourth point (with branch coverage 64.74%). This can lower down the FDP for the range represented by the third point, and thus affect the whole graph's flow as well. Data from Table II show that the correlation coefficients are greater than 0.5 for both line and branch coverage for TCAS. In general, the data of coverage and FDP are positively correlated but the strength of the correlation is not very strong.

For KNAPSACK, from Figs. 9 and 10, we can observe a good exponential increase for fault detection probability against both line and branch coverage data. Correlation coefficients from Table II also state the strong correlation between the coverage (line and branch) and FDP for KNAPSACK program.

Based on the experimental results, we can say that the code coverage attained by MRs is a good indicator for the fault-detection effectiveness, but not the only one. There may present several other factors that have an impact on the effectiveness of MRs. For example, the structure of the subject program must also be considered. In our study, the value range of the coverage percentages on TCAS is much broader than that on KNAPSACK. For TCAS, there can be some situations where some program segments are covered by some MRs with low coverage, but not by those with high coverage. If faults are located in these segments, some MRs will not able to detect them, even if they have high coverage. In other words, when the coverage achieved by an MR disperses in a broad range from a very low value, we cannot guarantee that high code coverage always brings high fault-detection effectiveness.

In summary, there exists a correlation between the code coverage and the fault-detection effectiveness. In other words, execution behavior caused by the MR can be a very good estimator of MR's effectiveness. However, it is also necessary to consider other factors, such as the program structure when identifying the MRs.

V. THREATS TO VALIDITY

The threats to validity in our works are discussed as follows.

The internal validity of our study lies in the implementation of our experiment. Some errors might exist when executing the processes of test case generation and test output verification. However, these processes only involve some simple programming tasks. Moreover, the source code has been checked by different individuals.

The main concern about the external validity of our study exists in the subject programs. We have chosen two subject programs from two different platforms. TCAS is written in C which is a procedural language; on the other hand KNAPSACK is written in object oriented programming language Java. The purpose of choosing these two programs is to make this study platform independent. In order to make

this study scale independent, automated mutant generation tools were used to avoid any bias, which may be introduced by hand seeded fault. Moreover, the MRs ~~are~~ were identified by the testers, so such an identification process is subjective. In our study, we asked independent individuals to identify the MRs without a prior knowledge of the research question of the paper. This avoided us subconsciously identifying the MRs that favor our rationale.

The construct validity of this study comes with the measurement metrics used in the experiment. We have used two kinds of coverage criteria, namely line and branch coverage to measure the coverage percentages and thus reflect the execution behaviors caused by MRs. These two criteria are very basic code coverage criteria and they have been popularly used in practice. We measured the testing effectiveness of MRs based on their fault-detection effectiveness, which is also widely used in the community.

VI. Conclusion

Metamorphic testing technique alleviates the oracle problem by using a set of metamorphic relations (MRs). Many MRs can be identified on the basis of the algorithm or the logic of the software under test. Different MRs have different effectiveness for the detection of various faults. We can save both time and resources while testing by exploring MRs with high effectiveness. Some researchers have argued that the MRs that can cause different execution behaviors of the software under test will have a high effectiveness. In this paper, we conducted a case study to systematically investigate the relationship between the execution behaviors and the effectiveness of MRs. The code coverage achieved by the MRs is considered as the indicator of execution behavior caused by the MR.

An on-board aircraft conflict detection and resolution system (TCAS) and a program for solving the multiple knapsack problem (KNAPSACK) were selected as the subject programs for our study. In total, fourteen MRs and ten MRs are identified for these two programs, respectively. 10,000 source test cases are generated randomly and 10,000 follow-up test cases are generated according to each MR. Mutation analysis was conducted to evaluate the fault detection effectiveness of different MRs. Coverage data were collected and analyzed to find out the relationship between the execution behaviors and effectiveness of MRs.

It was found that MRs with low coverage have low effectiveness in detecting software faults. On the other hand a high coverage shows better performance in most cases. However high coverage does not necessarily imply a high effectiveness all time. Our experimental results showed that some MRs with high coverage cannot detect a large number of faults. Such an observation is also understandable, as an MR cannot detect a fault as long as the MR does not execute the statement containing that fault, even if the MR achieves high coverage. In a word, high coverage indicating diverse execution behavior is a good estimator of fault effectiveness for MR, but the high coverage value cannot be a perfect indicator for the fault detection effectiveness.

It is necessary to further investigate other factors that may affect the fault-detection effectiveness besides the code coverage. Another future work is to conduct more studies with various subject programs. We have used only line and branch coverage as code coverage criteria in this study. Other coverage criteria, such as data-flow criteria, and some inner inspections in the execution of the programs are also our future projects.

Acknowledgement

We are thankful to Shengqiong Wang and Ling Chen for their contributions in the preliminary experiment of this study, and Peishi Yong for helping in data collection. This project is supported by the Australian Research Council Discovery Project (ARC DP0984760).

References

[1] Binder, R. V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison Wesley, Massachusetts (2000).

[2] Briand, L. C., Penta, M. Di, Labiche, Y.: Assessing and improving state-based class testing: a series of experiments. IEEE Transactions on Software Engineering, 30 (11): 770–783(2004).

[3] CodeCover: an open source glass- box testing tool, http://codecover.or g/index.html.

[4] Chan, W. K., Chen, T. Y., Lu, H., Tse, T. H., Yau, S. S.: A metamorphic approach to integration testing of contextsensitive middleware-based applications. In Proceedings of the 5th International Conference on Quality Software (QSIC 2005). 241–249 (2005).

[5] Chan, W. K., Cheung, S. C., Ho, J. C. F., Tse, T. H.: Reference models and automatic oracles for the testing of mesh simplification software for graphics rendering. In Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006), 429–438 (2006).

[6] Chan, W. K., Cheung, S. C., Ho, J. C. F., Tse, T. H.: PAT: a pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. Journal of Systems and Software, Vol. 82, 422-434 (2008).

[7] Chen, T. Y., Cheung, S. C., Yiu, S. M.: Metamorphic testing: a new approach for generating next test cases.Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology (1998).

[8] Chen, T. Y., Feng, J., Tse, T. H.: Metamorphic testing of programs on partial differential equations: A case study. In Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2002). 327–333 (2002).

[9] Chen, T. Y., Ho., J. W. K., Liu, H., Xie, X.: "An innovative approach for testing bioinformatics programs using metamorphic testing". BMC Bioinformatics. 10-24 (2009).

[10] Chen, T. Y., Huang, D. H., Tse, T. H., Zhou, Z. Q.: Case studies on the selection of useful relations in metamorphic testing. In Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering. 569–583 (2004).

[11] Chen, T. Y., Tse, T. H., Zhou, Z. Q.: Semi-proving: An integrated method based on global symbolic evaluation and metamorphic testing. In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002). 191–195 (2002).

[12] Chen, T. Y., Tse, T. H., Zhou, Z. Quan.: "Fault-based testing without the need of oracles." Information and Software Technology. Vol. 45(1). 1-9 (2003)

[13] Davis, M. D., Weyuker, E. J.: Pseudo-oracles for non-testable programs. In Proceedings of the ACM '81 Conference. 254-257(1981).

[14] Gotlieb, A., Botella, B.: Automated metamorphic testing. In Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC2003). 34–40 (2003).

[15] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T.: Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), 191–200 (1994).

[16] Jia, Y., Harman, M.: A Custimizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. The 3rd Testing Academia and Industry Conferrence- Practice and Research Techniqes TAIC PART'08, Windsor, UK, 29th-31st August 2008.

[17] Kaner, C. Center for Software Testing & Research: Examples of Test Oracle.http://www.testingeducation.org/k04/OracleExamples.htm.

[18] Kenney, J. F. and Keeping, E. S. Mathematics of Statistics, Pt. 1, 3rd ed. Princeton, NJ: Van Nostrand, pp. 25-26, 1962.

[19] Lau, H. T. 2007, A java library of graph algorithms and optimisation, Taylor & Francis Group, Boca Raton.

[20] LCOV- the LTP GCOVextension,http://ltp.sourceforge.net/coverage/lcov.php

[21] Gcov-kernel - a gcov infrastructure for the Linux kernel, http://ltp.source forge .net/coverage/gcov.php

[22] Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: An Automated Class Mutation System. Journal of Software Testing, Verification and Reliability. 15(2):97-133, June 2005.

[23] Manolache, L. I., Kourie, D.G.: Software testing using model programs. Software: Practice and Experience. Vol. 31, 1211–1236, (2001).

[24] Mayer, J., Guderlei, R.: Test oracles using statistical methods. In Lecture Notes in Informatics P-58, 179–189 (2004).

[25] Mayer, J., Guderlei, R.: An Empirical Study on the Selection of Good Metamorphic Relations. In Proceedings of the 30th Annual international Computer Software and Applications Conference (COMPSAC 2006). 475-484 (2006).

[26] Meyer, B.: Eiffel: the Language. Prentice Hall (1992).

[27] Murphy, C., Shen, K., Kaiser, G.: Automatic system testing of programs without test oracles. In Proceedings of the Eighteenth international Symposium on Software Testing and Analysis (ISSTA 2009). 189-200 (2009).

[28] Tse, T. H., Yau, S. S., Chan, W. K., Lu, H., Chen, T. Y.: Testing context-sensitive middleware-based software applications. In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004). 458–466 (2004).

APPENDIX

A. MRs of TCAS

We have identified 14 MRs for TCAS, as described below. In the following, source and follow-up test cases are denoted by T_s and T_f, respectively, while the test outputs of T_s and T_f are denoted by O_s and O_f, respectively.

- MR1: Given that the intruder aircraft does not have the TCAS system, if T_s and T_f only differ in whether the intruder aircraft has an intention or not, we should have the relation $O_f = O_s$.
- MR2: Given that the intruder aircraft does not have the TCAS system, if T_s and T_f only differ in whether the report describing the presence of any intruder is valid or not, we should have the relation $O_f = O_s$.
- MR3: Given that the intruder aircraft does not have any intention, and the report describing the presence of any intruder is valid, if T_s and T_f only differ in whether the intruder aircraft has the TCAS system or not, we should have the relation $O_f = O_s$.

The next eight relations (MR4-MR11) have an additional prerequisite that includes the following conditions.

(1) The TCAS system on the controlled aircraft has a high confidence, **and**
(2) The vertical converging speed is not larger than 600, **and**
(3) The current vertical separation between the two aircrafts at the closest point will be larger than 600 if the controlled aircraft maintains its trajectory, **and**
(4) (i) The intruder aircraft does not have the TCAS system,
or
(ii.a) the intruder aircraft does not have any intention,
and
(ii.b) the report describing the presence of any intruder is valid.

- MR4: Given that the current altitude of the controlled aircraft is smaller than that of the intruder aircraft, and the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward maneuver, if T_s and T_f differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward maneuver, we should have the relation $O_f \neq O_s$.
- MR5: Given that the current altitude of the controlled aircraft is not smaller than that of the intruder aircraft, and the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled aircraft initiates an upward maneuver, if T_s and T_f differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward maneuver, we should have the relation $O_f \neq O_s$.
- MR6: Given that the vertical separation between two aircrafts will be no larger than the calculated inhibit biased climb and smaller than the threshold value if the controlled aircraft initiates an downward maneuver, if T_s and T_f differ in the relation between the current altitudes of the two aircrafts, we should have the relation $O_f \neq O_s$.
- MR7: Given that the vertical separation between two aircrafts will not be smaller than the calculated inhibit biased climb if the controlled aircraft initiates a downward maneuver and not larger than the threshold value if the controlled aircraft initiates an upward

maneuver, if T_s and T_f differ in the relation between the current altitudes of the two aircrafts, we should have the relation $O_f \neq O_s$.

- MR8: Given that the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward maneuver, if T_s and T_f differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward maneuver, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR9: Given that the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled aircraft initiates an upward maneuver, if T_s and T_f differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward maneuver, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR10: Given that the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward maneuver, if T_s and T_f differ in the relation between the current altitudes of the two aircrafts, we should have the relation: if $O_s = 0$, $O_f \notin \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR11: Given that the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled aircraft initiates an upward maneuver, if T_s and T_f differ in the relation between the current altitudes of the two aircrafts, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR12: Given that other parameters can be randomly changed, if T_s and T_f differ in whether the TCAS system on the controlled aircraft has a high confidence or not, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR13: Given that other parameters can be randomly changed, if T_s and T_f differ in whether the vertical converging speed is larger than 600 or not, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

- MR14: Given that other parameters can be randomly changed, if T_s and T_f differ in whether the current vertical separation between the two aircrafts at the closest point will be larger than 600 or not where the controlled aircraft maintains its trajectory, we should have the relation: if $O_s = 0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

B. MRs of KNAPSACK

Ten MRs were identified for KNAPSACK as follows. In the following, the source test case is denoted as $T = \{P, W, C\}$, where $P = \{p_1, p_2, \ldots, p_n\}$, $W = \{w_1, w_2, \ldots, w_n\}$, and $C = \{c_1, c_2, \ldots, c_m\}$. The output of the source test case is denoted as $O = \{Y, TP\}$, where $Y = \{y_1, y_2, \ldots, y_n\}$, and TP is a positive integer representing the total profit.

- MR1: Swap the k^{th} and the l^{th} items, where $1 \leq k < l \leq n$, and $p_k \neq p_l$ or $w_k \neq w_l$. We can get the follow-up test case $T' = \{P', W', C\}$, where $P' = \{p_1, p_2, \ldots, p_l, \ldots, p_k, \ldots, p_n\}$ and $W' = \{w_1, w_2, \ldots, w_l, \ldots, w_k, \ldots, w_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = \{y_1, y_2, \ldots, y_l, \ldots, y_k, \ldots, y_n\}$ and $TP' = TP$.

- MR2: Select the k^{th} item where $y_k = 1$ (that is, the k^{th} item is put into the 1^{st} knapsack), and then increase its profit by a positive integer c, that is, $p'_k = p_k + c$. We can get the follow-up test case $T' = \{P', W, C\}$, where $P' = \{p_1, p_2, \ldots, p'_k, \ldots, p_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$, where $Y' = \{y'_1, y'_2, \ldots, y'_n\}$. We should have $y_j \bullet y'_j = 0$ iff $y_j = y'_j = 0$, and $TP' = TP + c$.

- MR3: Select the k^{th} item where $y_k = 0$ (that is, the k^{th} item is not put into any knapsack), and then increase its weight by a positive integer c, that is, $w'_k = w_k + c$. We can get the follow-up test case $T' = \{P, W', C\}$, where $W' = \{w_1, w_2, \ldots, w'_k, \ldots, w_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$, where $Y' = \{y'_1, y'_2, \ldots, y'_n\}$. We should have $y_j \bullet y'_j = 0$ iff $y_j = y'_j = 0$, and $TP' = TP$.

- MR4: Select the k^{th} item where $y_k = 0$ (that is, the k^{th} item is not put into any knapsack), and then decrease its profit by a positive integer c, that is, $p'_k = p_k - c$. We can get the follow-up test case $T' = \{P', W, C\}$ where $P' = \{p_1, p_2, \ldots, p'_k, \ldots, p_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$, where $Y' = \{y'_1, y'_2, \ldots, y'_n\}$. We should have $y_j \bullet y'_j = 0$ iff $y_j = y'_j = 0$, and $TP' = TP$.

- MR5: Change the capacity of the 1^{st} knapsack to a new value c'_1, where c'_1 is equal to the summary of the weights of all items put into the 1^{st} knapsack. We can get the follow-up test case $T' = \{P, W, C'\}$ where $C' = \{c'_1, c_2, \ldots, c_m\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = Y$ and $TP' = TP$.

- MR6: Add a new item at the position $n + 1$, where $p_{n+1} = \min(p_j)$ and $w_{n+1} = \max(w_j)$ for all j such that $y_j \neq 0$. We get the follow-up test case $T' = \{P', W', C\}$, where $P' = \{p_1, p_2, \ldots, p_n, p_{n+1}\}$ and $W' = \{w_1, w_2, \ldots, w_n, w_{n+1}\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = \{y_1, y_2, \ldots, y_n, 0\}$ and $TP' = TP$.

- MR7: Select the k^{th} item where $y_k = 0$ (that is, the k^{th} item is not put into any knapsack), and then delete it. We can get the follow-up test case $T' = \{P', W', C\}$, where $P' = \{p_1, p_2, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n\}$ and $W' = \{w_1, w_2, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$, where $Y' = \{y'_1, y'_2, \ldots, y'_{k-1}, y'_{k+1}, \ldots, y'_n\}$. We should have $y_j \bullet y'_j = 0$ iff $y_j = y'_j = 0$ ($j \neq k$), and $TP' = TP$.

- MR8: Select the k^{th} item where $y_k = 1$ (that is, the k^{th} item is put into the 1^{st} knapsack), delete it, and then decrease the capacity of the 1^{st} knapsack by w_k, that is, $c'_1 = c_1 -$

w_k. We can get the follow-up test case T' = {P', W', C'}, where P' = {p_1, p_2, …, p_{k-1}, p_{k+1}, …, p_n},W' = {w_1, w_2, …, w_{k-1}, w_{k+1}, …, w_n}and C' = {$c_1 - w_k$, c_2, …, c_m}. The output corresponding to T' is O' = {Y', TP'}, where Y' = {y'_1, y'_2, …, y'_{k-1}, y'_{k+1}, …, y'_n}. We should have $y_j \bullet y'_j$ = 0 iff $y_j = y'_j = 0$ ($j \neq k$), and TP' = TP-p_k.

- MR9: Select the k^{th} and l^{th} items where $1 \leq k < l \leq n$ and $y_k = y_l = i \neq 0$, delete the l^{th} item, and then create a new k^{th}, where $p'_k = p_k + p_l$ and $w'_k = w_k + w_l$. We can get the follow-up test case T' = {P', W', C}, where P' = {p_1, p_2, …, p_k+p_l, …, p_{l-1}, p_{l+1},…, p_n} and W' = {w_1, w_2, …, w_k+w_l, …, w_{l-1}, w_{l+1},…, w_n}. The output corresponding to T' is O' = {Y', TP'}. We should have TP' = TP.

- MR10: Delete all items put into the 1st knapsack and delete the 1st knapsack. Given that η items were in the 1st knapsack and their total profit is ν, we can get the follow-up test case T' = {P', W', C'}, where P' = {p'_1, p'_2, …, $p'_{n-\eta}$}, W' = {w'_1, w'_2, …, $w'_{n-\eta}$}, and C' = {c_2, c_3, …, c_m}. The output corresponding to T' is O' = {Y', TP'}. We should have TP'=$TP - \nu$.