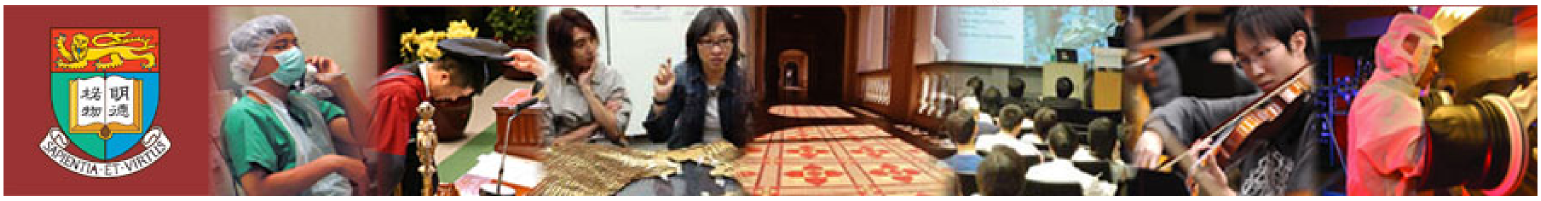


Piping Classification to Metamorphic Testing: An Empirical Study towards Better Effectiveness for the Identification of Failures in Mesh Simplification Programs [*]

W. K. Chan
City Univ. of Hong Kong
wkchan@cs.cityu.edu.hk

Jeffrey C. F. Ho
University College London
jcfho@cs.hku.hk

T. H. Tse [†]
The Univ. of Hong Kong
thtse@cs.hku.hk

Abstract

Mesh simplification is a mainstream technique to render graphics responsively in modern graphical software. However, the graphical nature of the output poses a test oracle problem in testing. Previous work uses pattern classification to identify failures. Although such an approach may be promising, it may conservatively mark the test result of a failure-causing test case as passed.

This paper proposes a methodology that pipes the test cases marked as passed by the pattern classification component to a metamorphic testing component to look for missed failures. The empirical study uses three simple and general metamorphic relations as subjects, and the experimental results show a 10 percent improvement of effectiveness in the identification of failures.

Keywords: *test oracle problem, mesh simplification, metamorphic testing, classification.*

1. Introduction

Testing is essential to assure the quality of software applications. Software testers may identify test cases for a program using test data selection techniques such as control flow testing, dataflow testing, random testing, adaptive random testing, and fault-based testing. After executions of the test cases, test outputs are checked to determine whether or not they reveal any failure. A *test oracle* is the mechanism against which testers can check the output of a program and decide whether it is correct. When the test oracle is not available or is too expensive to evaluate, it is referred to as a *test oracle problem*. Various techniques such as golden version [5], reference model [11, 12], assertion checking [7, 29], and metamorphic testing [8, 9, 13, 14, 20] have been proposed to alleviate the test oracle problem.

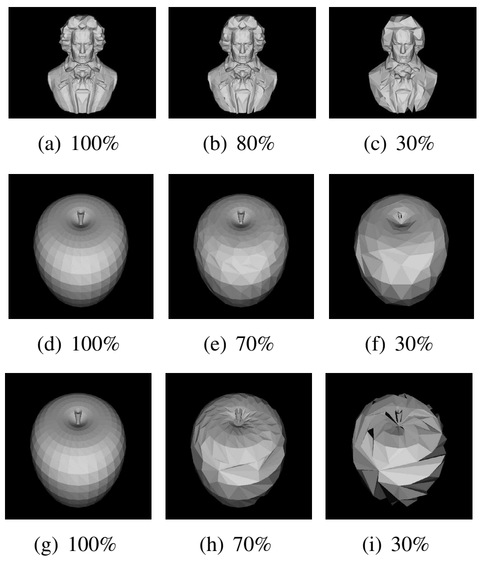


Figure 1. Mesh simplification of polygonal models of a Beethoven statue, a properly rendered apple, and a badly rendered apple. From [11].

Content-rich software such as multimedia applications and graphic rendering applications is undoubtedly an important class of modern software. They usually accept media specifications or objects [1] and then render the required graphics. For real-life interactive graphics-based software such as medical imaging [1] and graphics-based entertainment, slow rendering of graphics is intolerable. A mainstream technique to ease the process is *mesh simplification* [15, 22, 23], which

[*] This research is supported in part by a grant of the Research Grants Council of Hong Kong (project no. 714504), a grant of City University of Hong Kong (project no. 7200079), and a grant of The University of Hong Kong.

[†] All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2557 8447. Email: thtse@cs.hku.hk.

[1] In such formats as MRI files, X3D files, and MPEG files.

converts a given three-dimensional (3D) polygonal model to one with fewer polygons while appearing to be similar to the original. Figure 1 shows a Beethoven statue and two apples being modeled by different number of polygons via mesh simplification.

Since rendering is often computationally expensive and dependent on the platform where the process takes place, precise expected results are hard to be specified in advance. A number of preliminary experiments have been conducted on various approaches to tackle the test oracle problem: Hu et al. [20] have statistically shown that metamorphic testing is more effective than asserting checking in the context of open-source object-oriented software. Chan et al. [10] have demonstrated the feasibility of using a supervised machine learning approach to identify failures in multimedia applications, where the training information is extracted from the implementation under test and some of its test cases. Nevertheless, such an approach requires that some of the test results can be determined in advance. An alternate way is to use a reference model to provide the training information. Chan et al. [11] have conducted experiments to show that a resembling rendering approach is better than a dissimilar one when serving as a pseudo-oracle. Chan et al. [12] have further shown that, when compared with a simple reference model, the use of a resembling but sophisticated reference model is more effective and accurate but less robust. Owing to the conservative nature of the training, however, many failure-causing test cases are classified as *passed* test cases. This hinders the effectiveness of the proposal in [11, 12].

Many techniques, such as assertion checking, directly verify the output of a test case. Metamorphic testing (MT) verifies relationship among the outputs of a collection of test cases instead [13]. Given a test case (known as a *source* test case in MT), testers may be unable to directly verify whether the output is correct. To tackle the test oracle problem, MT recommends the construction of *follow-up* test cases. In the simplest form, MT compares the test outputs of both the source and follow-up test cases to check whether they may contradict any necessary relation, thus indicating a failure.

The main contributions of this paper are two-fold: It presents a methodology to integrate analytical and statistical ways of identifying failures in the testing of mesh simplification programs. It also reports the first set of evaluation results of its type.

The rest of the paper is organized as follows: Section 2 reviews related work on the testing of software with graphical interfaces. A review of metamorphic testing is presented in Section 3. Section 4 presents our methodology. Section 5 presents our experimental setup, results, and threats to validity. Finally, Section 6 concludes the paper.

2. Related Work

We review related work that uses machine learning approaches as pseudo-oracles, as well as related work on metamorphic testing and other approaches to alleviating the test oracle problem. For brevity, we shall focus on the testing of software with graphical interfaces.

Berstel et al. [3] design the VEG language to describe graphical user interfaces and show that model checkers may verify properties against a specification written in VEG without referring to the source program. Our approach does not rely on source code either, but our work involves dynamic analysis whereas theirs is static. D'Ausbourg et al. [16] support the formal design of operations in user interface systems via a software environment. The formal design can be verified as in [3]. Memon et al. [28] use a test specification of internal object interactions as a means of detecting inconsistencies between the test specification and the resulting execution sequence of events for each test case. This approach is often used in the conformance testing of telecommunication protocols. Sun et al. [33] propose a similar approach for test harnesses. Memon et al. [27] further evaluate several types of pseudo-oracle for GUIs. Their results suggest the use of simple pseudo-oracles for large test sets and complex pseudo-oracles for small test sets. Our work does not explore such kinds of tradeoff, but integrates a complex pseudo-oracle (classification) with a simple one (metamorphic relation).

There are other approaches to test programs with graphical outputs. gDEBugger[2] checks the conformance of the list of commands issued by an application to the underlying graphics-rendering Application Programming Interface (API) of OpenGL [32]. As explained in [4, 11], however, many different sets of commands may be rendering the same graphical image. To test programs with interfaces with virtual reality applications, Bierbaum [4] proposes a framework to record selected intermediate states of the program for given test cases and contrast them against the expected ones. Following Chan et al. [11], we do not use the internal states of the program under test. Mayer [25] proposes to use explicit statistical formulas such as mean and distributions to determine whether the output exhibits the same characteristics. Following [11], we use classifiers to handle such features instead.

[2] Available at http://www.gremedy.com/.

The test oracle problem has also been studied in other contexts. Ostrand et al. [30] propose an integrated environment so that testers can easily review and modify their test scripts. Dillon and Ramakrishna [17] prune the search space of test oracles constructed from a specification. Baresi et al. [2] add assertions [29] to programs to check their intermediate states.

More specifically, there are techniques for applying pattern classifications to alleviate the test oracle problem. Last and others [21, 34] train a classifier to augment the incomplete specification of legacy systems, treating the latter as a golden version. As we have explained, golden versions are often not available for rendering-intensive software. Podgurski et al. and their research group classify failure cases into categories by machine learning [31] and then refine the categories using the classification tree technique [18]. Bowring et al. [6] apply machine learning to regression testing of a consecutive sequence of minor revisions of the same program to identify failures in versions. Their approach is similar to the reference model approach by Chan et al. [11]. However, the approach by Bowring et al. requires the source code of the program under test while that by Chan et al. does not. Another pattern classification approach by Chan et al. [10] does not use reference models.

3. Metamorphic Testing

This section revisits metamorphic testing (MT). The central idea of MT is to check necessary properties that relate multiple test cases and their results with a view to revealing failures. Such necessary properties are known as metamorphic relations.

A *metamorphic relation* (*MR*) [9, 14] is a relation over a set of distinct inputs and their corresponding outputs of the target function f to be implemented by the program P under test. For the sine function, for instance, given any inputs x_1 and x_2 such that $x_1 + x_2 = \pi$, we must have $\sin x_1 = \sin x_2$.

Given a test case x_1, using the above MR for the sine function, a tester will construct a follow-up test case x_2 based on the relation $x_1 + x_2 = \pi$. By executing the program p over both x_1 and x_2, the tester will obtain the respective test results y_1 and y_2 and then check whether y_1 is equal to y_2 using the relation $\sin x_1 = \sin x_2$. If the equality is breached, MT detects a failure.

4. Methodology

This section proposes a testing methodology that integrates the pattern classification technique and metamorphic testing. It investigates the integration of statistical and analytical techniques for alleviating the test oracle problem.

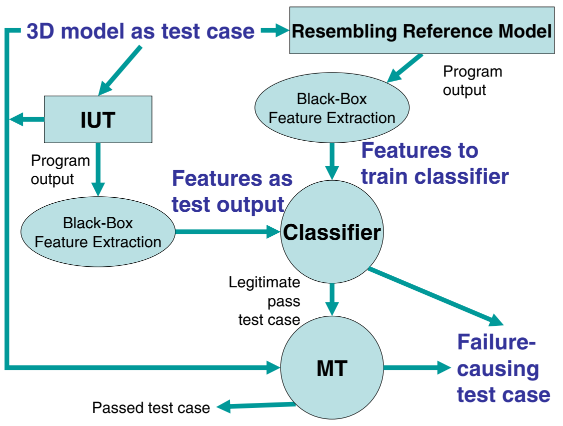

3D model as test case
Resembling Reference Model
IUT
Program output
Black-Box Feature Extraction
Program output
Black-Box Feature Extraction
Features to train classifier
Features as test output
Classifier
Legitimate pass test case
MT
Failure-causing test case
Passed test case

Figure 2. Blueprint of the methodology.

In our methodology, a trained classifier may label a test case as *failed* or *passed*. A test case marked as *failed* would catch the attention of testers. Because of the statistical nature of a classifier, however, test outputs marked as *passed* may still be failures. Thus, even after a statistical classifier has determined that a (source) test case does not reveal any failure, our methodology proposes to pipe the test case (and its test output) to an analytical MT component for further checking. Figure 2 depicts the blueprint of the methodology.

In this way, the integrated strategy saves checking efforts in MT for the test cases that have been classified as *failed*. Still, it is not known how many of the test cases classified by the statistical classifier as *passed* are, in fact, failure-causing. Will the extra step of applying MT be worth the effort? A research question thus arises: *During the testing of mesh-simplification software, how much improvement in the effectiveness of failure identification will result by piping the results of the machine learning approach to MT?*

Since the pattern classification step relevant to the methodology has already been evaluated preliminarily by Chan et al. [11], our current experiments, to be presented in the next section, will focus on evaluating the second step of our methodology, which builds atop the previous experiments in [11]. We identify simple metamorphic relations for the MT component proposed above. Based on these metamorphic relations, the MT approach alone can identify 30% of the failures. Using 10% data as training samples to train the classifier, an initial study of our methodology shows a 10 per-cent boost in the effectiveness of correctly identifying program failures from the test outputs. The details of the empirical study are presented in the next section.

5. Empirical Study

In this section, we describe the setup and results of our empirical study. The study is built on top of previous experiments by Chan et al. [11]. The entire classification data of [11] will be used as a starting point. Specifically, we use a few subject programs to construct mutants, and then execute them with open-source models to generate a test pool. A small number of the test cases in the pool are used to train the classifier on the reference model. We then apply the remaining test cases to the program under test and ask the trained classifier to mark them as *passed* or *failed*. Apart from this, we also design a few metamorphic relations. We then pipe the test cases marked as *passed* by the classifier to the metamorphic testing process. Further details of the classification process can be found in [11].

5.1. Subject Programs

For the purpose of comparison, we use the same Java programs studied in [11] as subject programs. Each of the four programs has a distinct mesh simplification algorithm, namely a shortest edge algorithm in *Shortest*, Melax's simplification algorithm [26] in *Melax*, a quadric algorithm [19] in *Quadric*, and a quadric algorithm weighted by the area of triangles [19] in *QuadricTri*. *Shortest* is a simple and direct mesh simplification algorithm. It always picks the shortest edge of a mesh to collapse. *Melax* measures the cost of each edge as a product of its length and curvature, and iteratively picks the edges with the lowest costs to remove. *Quadric* contracts pairs of vertices rather than edges, so that unconnected regions can also be joined. It approximates contraction errors by quadric matrices. *QuadricTri* improves on *Quadric* by considering also the sizes of triangles around vertices during contraction. *Quadric* and *QuadricTri* are two resembling subject programs. A preliminary empirical study by Chan et al. [11] indicate that the use of a resembling reference model to train a classifier for failure identification is better than that of a dissimilar model. Hence, in our present experiment, we focus on the improvement in the effectiveness of failure identification between *Quadric* and *QuadricTri*.

Each program accepts two inputs: a 3D polygonal model file in standard .PLY format and an integer (from 0 to 100) indicating the target percentage of polygons that will remain after mesh simplification. If the value of the input integer is zero, for instance, only the background will be shown. The backgrounds of all outputs are black in color. Each program fits the 3D polygonal model in a bounding box between $(-1, -1, -1)$ and $(1, 1, 1)$, centered at $(0, 0, 0)$. The image resolution is standardized to 800 pixels $\times$ 600 pixels.

5.2. Test Cases

Chan et al. [11] use a supervised machine learning approach to classify test cases into two categories: *passed* and *failed*. We apply the same classification procedure in our empirical study.

To collect training samples for the *passed* class, we execute a set of 44 3D polygonal models[3] with up to 17,000 polygons in each reference system. We do not use the remaining 8 available models because they are outliners in terms of the number of polygons they contain. In order to better utilize the 3D polygonal models, we rotate each model in 22 different orientations, and each orientation is further used to construct 11 images at various simplification ratios (from 0% to 100% with increments of 10%). In other words, $44 \times 22 \times 11 = 10,648$ images are produced.

To collect training samples for the *failed* class, program mutants are generated from the reference system using a mutation tool known as MuJava [24]. Screening measures have been taken by Chan et al. [11] to ensure that the mutants generated are useful and non-equivalent. A total of 3,060 mutants remain, as shown in Table 1. We use all of these mutants to collect the features defined in [11] through a total of more than 440,000 program executions.

Shortest	*Melax*	*Quadric*	*QuadricTri*
350	401	1,122	1,187

Table 1. Numbers of mutants used.

5.3. Classification Phase

After capturing the test cases and features, an experiment in the style of [11] is conducted for pattern classification using the C4.5 classifier. It classifies test cases into two categories: *passed* and *failed*.

The main focus of the present paper is to use the *passed* test cases and pipe them to the next metamorphic testing phase (as described in Section 5.4), building atop the finding in [11] that testers are better off using a resembling reference model to train a classifier. We have experimented with the use of one to five models (representing 2.2% to 11.4% of the total of 44 models) to train the classifier and classify all the other 10,000 test cases.

The mean classification results are depicted in Figure 3. It shows the results of classifying the test

[3] Available at http://www.melax.com/polychop/lod_demo.zip. According to this source, they are "big demo[s]" to convince skeptical visitors.

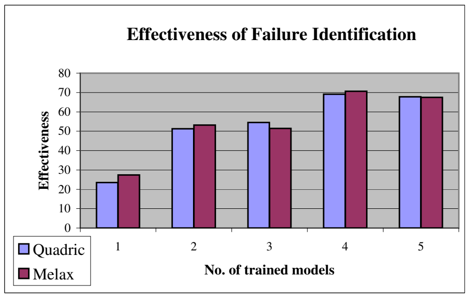


Figure 3. Classification results by the C4.5 classifier trained by *QuadricTri* as the reference model.

cases for *Quadric* and *Melax* after the classifier has been trained on *QuadricTri* as the reference model. Using "approximately 10 percent of the data" as a sample training criterion, we may focus on the rightmost two pairs of bars. They represent the situations where the classifier has been trained with 4 and 5 models, respectively, corresponding to 9.1% and 11.4% of the dataset. The *y*-axis represents the *effectiveness*, which is the percentage of failure-causing test cases that are correctly marked as *failed*. The bars for *Quadric* are 69.1% and 67.79%, respectively, while those for *Melax* are 70.69% and 67.53%, respectively. We shall use them as benchmarks for evaluating the improvements due to the metamorphic testing phase.

According to Figure 3, the classification results of *Melax* are at least as good as those of *Quadric*, contradicting the finding in [11] that "the reference should preferably be a resembling system that the new implementation improves on". We investigate further about the issue in [12]. On closer look, we find that in the case of *Melax*, many test cases that are not failure-causing are classified as *failed*, making the quality of classification phase unsatisfactory. In the case of *Quadric*, the vast majority of these test cases are retained in the *passed* category. Figure 4 shows the percentage of non-failure-causing test cases that are correctly marked as *passed*. We refer to this percentage as the *robustness*. It can be observed that all the bars for *Quadric* are much higher than those for *Melax*. We have analyzed the results of classifying test cases into *failed* and *passed* categories using a resembling reference model. They show that the approach is conservative, in the sense that it has a high score in robustness. This would avoid misclassifying a *passed* test case as *failed* and, hence, save the testers' effort in futile attempts to debug such test cases.

Because of the conservative bias, however, *failed* test cases may be grouped under the *passed* category and ignored by testers. We propose to further mine them out using metamorphic testing as explained in the next section.

5.4. Metamorphic Testing Phase

As we have explained in Section 1 and reported in Section 5.3 as well as in [11], the classification phase may group *failed* test cases into the *passed* category. In this phase, we aim at digging out the failure-causing test cases from the pool of test cases that have been marked as *passed* by the classification phase. We first describe the metamorphic relations for the experiments, and then report on the findings.

Metamorphic Relations: We acknowledge the real-life situation where the complexity of interactions of features in a program makes the definition of a complex metamorphic relation tedious and effortful. We propose to use three simple metamorphic relations to check the *passed* test cases. As defining an adequate set of metamorphic relations is still an open problem, we pick the following generic metamorphic relations based on a general understanding of mesh simplification, so that they are not tied to any particular simplification strategy. At the same time, we are full aware of the implication that these MRs may be weak.

- The first metamorphic relation checks the size of the bounding box rendered from a source test case against that from the non-simplified 3D polygonal model (that is, when the input integer = 100). This is akin to a common practice in assertion checking that the size of the bounding box is verified after each iteration.

- The second MR reverses the order of the vertexes of polygons in the .PLY input file. It is analogous to requesting the program to render the graphic in the reversed direction.

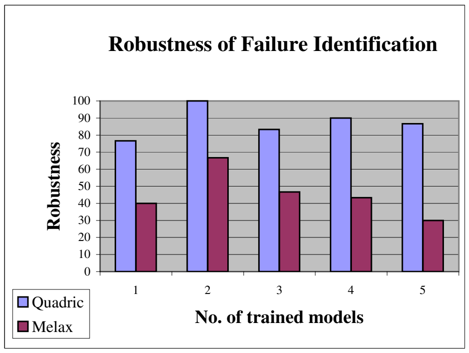


Figure 4. Robustness of retaining non-failure-causing test cases in the *passed* category.

- The third one changes the input so that the rendered image *should be* upside down.

In the experiments, the first relation is further divided into two minor versions, and so is the second relation. Since the capabilities of failure identification only differ marginally, we shall not further describe these minor versions. For the sake of brevity, we shall not describe the implementation of our algorithm to exercise MT using these relations.

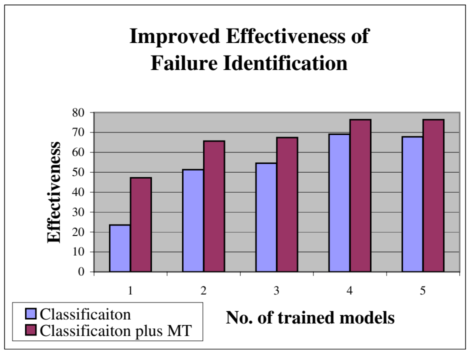
Let us firstly estimate the overall capability of failure identification by these MRs. Because of the sheer size of the test pool used in the experiments, we cannot do the estimate by applying the MRs exhaustively to all the test cases. We randomly select a subset as source test cases. Using our implementation for exercising MT, we construct follow-up test cases and then determine whether failures are revealed. The results show that 29.4% of the failure-causing test cases of *Melax* and 34.1% for *Quadric* are detected.

Readers may express a concern that these metamorphic relations appear to be weak. This, in fact, makes the research question more interesting: *Can the effectiveness of failure identification be improved by piping results of the machine learning approach to a number of weak MRs?*

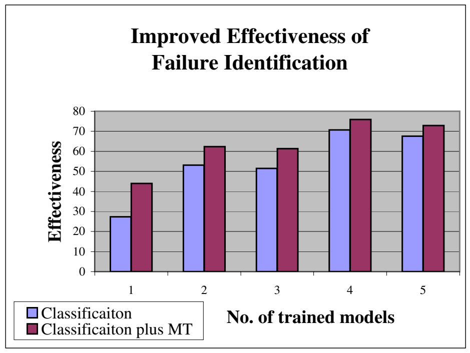
Piping: We apply all the test cases that have been identified as *passed* in the classification phase to the metamorphic testing implementation. We then count the number of test cases that fail in one or more of the metamorphic relations. The improvements in effectiveness for both *Quadric* and *Melax* are shown in Figure 5. The interpretation of the *x*- and *y*-axes is the same as that for Figure 3.

Using the popular 80-20 rule as the yardstick for adequate robustness (see Figure 4), we may ignore the leftmost pairs of bars in Figure 5(a). We then compute the improvements in effectiveness by subtracting the "Classification" values from the corresponding "Classification plus MT" values. The improvements are, from left to right, 14.38%, 12.9%, 7.33%, and 8.61%. The mean improvement is 10.8%, which is encouraging.

Although we do *not* recommend the use of a dissimilar approach as a reference model, we have included a plot of the improvements in effectiveness for *Melax* in Figure 5(b) for the sake of comparison. For every pair of bars in Figures 5(a) and (b), let us concentrate on the improvement in effectiveness as indicated by the increase in height of the second bar over the first one. In all five corresponding pairs of bars in the two figures, the improvements for *Quadratic* are always more than that for *Malex*. The mean



(a) Quadric



(b) Malex

Figure 5. Improved effectiveness via the proposal.

improvement for the former is 13.4% and that for the latter is 9.2%. If we consider only the rightmost 4 pairs of bars in each plot, the means will be 10.8% and 7.4%, respectively.

In all the cases analyzed above, the use of a resembling reference system has a positive and statistically significant difference in both the classification phase and the integrated approach. Also, since the metamorphic testing approach can identify a large number of failures not detectable by the classification phase in all the analyzed cases, our preliminary results show that metamorphic testing complements the classification approach effectively. The results in [11, 12] and the present paper provide the first set of evaluation results of this type.

5.5. Threats to Validity

In this section, we discuss the threats to validity of our empirical study.

We are dealing with rendering-intensive software in our study. Because of the oracle problem in verifying graphical outputs, we use feature extraction techniques to tackle the issue instead of directly comparing the actual outputs. We realize from the machine learning community that the selection of useful features plays a central role in the effectiveness of a classifier. In our metamorphic testing phase, we still apply the same features as used in the classification phase to identify failures. Intuitively, using a different set of features may affect the results, be it in a positive or negative manner. To alleviate this threat, generic features such as the standard frequency spectrum are used in the study. This is in line with the philosophy in [11].

Our implementation uses openGL to render graphics. While openGL is a popular standard, there are other choices such as DirectX and Flash. We have only experimented with a few implementations of mesh simplification algorithms. There are many other rendering algorithms. The generalization of our proposal, therefore, warrants more research. Also, our work is built on top of the C4.5 classifier. While it is an extremely important and classical algorithm in data mining, there are other classifiers that can be used.

Our experiment is done on a set of 44 open-source 3D polygonal models. They include many different graphics including a chair, a spider, a teapot, a tennis shoe, a weathervane, a street lamp, a sandal, a cow, a Porsche car, an airplane, and so on. Naturally, they do not represent all types of 3D polygonal model.

The choice of the three metamorphic relations is based on our experience. We are aware that the quality of metamorphic relations is important. We deliberately use simple and coarse metamorphic relations in our study. The results of our experiments serve as a baseline for further investigations.

6. Conclusion

A lot of modern software uses graphics at various levels of detail to improve the user friendliness. Mesh simplification is a mainstream technique to help render graphics responsively. The graphic nature of the output, however, causes a test oracle problem when testing these programs. Previous pattern classification proposals assume that testers can train classifiers using the behaviors of the implementation under test. Our previous work recognizes the use of resembling reference models to guide the training phase. Because of the conservative nature of classifiers, many test cases classified into the passed category may, in fact, be failed ones, thus lowering the effectiveness in identifying failures.

In this paper, we propose a methodology that pipes pattern classification to metamorphic testing. It uses the metamorphic testing approach to further check test cases marked as passed by a classifier. We further report an empirical study that applies three simple, general and coarse metamorphic relations to produce follow-up test cases for metamorphic testing. The experimental results show a 10 percent improvement in effectiveness, which is encouraging. Future work includes the development of new techniques to filter out false positive cases in the failed category and a tighter integration of pattern classification and metamorphic testing.

References

[1] M. N. Ahmed, S. M. Yamany, N. Mohamed, A. A. Farag, and T. Moriarty. A modified fuzzy c-means algorithm for bias field estimation and segmentation of MRI data. *IEEE Transactions on Medical Imaging*, 21 (3): 193–199, 2002.

[2] L. Baresi, G. Denaro, L. Mainetti, and P. Paolini. Assertions to better specify the amazon bug. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (*SEKE 2002*), pages 585–592. ACM Press, New York, 2002.

[3] J. Berstel, S. C. Reghizzi, G. Roussel, and P. San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transactions on Software Engineering and Methodology*, 14 (2): 124–167, 2005.

[4] A. Bierbaum, P. Hartling, and C. Cruz-Neira. Automated testing of virtual reality application interfaces. In *Proceedings of the Eurographics Workshop on Virtual Environments* (*EGVE 2003*), pages 107–114. ACM Press, New York, 2003.

[5] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, Reading, Massachusetts, 2000.

[6] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2004*), *ACM SIGSOFT Software Engineering Notes*, 29 (4): 195–205, 2004.

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

[8] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering* (*SE 1998*), pages 191–197. ACTA Press, Calgary, Canada, 1998.

[9] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16 (5): 677–703, 2006.

[10] W. K. Chan, M. Y. Cheng, S. C. Cheung, and T. H. Tse. Automatic goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: an experimental case study. *Journal of*

Systems and Software, 79 (5): 602–612, 2006.

[11] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. 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)*, pages 429–438. IEEE Computer Society Press, Los Alamitos, California, 2006.

[12] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. PAT: a pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. Submitted for publication.

[13] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 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, Hong Kong, 1998.

[14] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. 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*), *ACM SIGSOFT Software Engineering Notes*, 27 (4): 191–195, 2002.

[15] P. Cignoni, C. Rocchini, and G. Impoco. A comparison of mesh simplification algorithms. *Computers and Graphics*, 22 (1): 37–54, 1998.

[16] B. d'Ausbourg, C. Seguin, G. Durrieu, and P. Roch. Helping the automated validation process of user interfaces systems. In *Proceedings of the 20th International Conference on Software Engineering* (*ICSE 1998*), pages 219–228. IEEE Computer Society Press, Los Alamitos, California, 1998.

[17] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* (*ISSTA 1996*), *ACM SIGSOFT Software Engineering Notes*, 21 (6): 106–117, 1996.

[18] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE 2004*), pages 451–462. IEEE Computer Society Press, Los Alamitos, California, 2004.

[19] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (*SIGGRAPH 1997*), pages 209–216. ACM Press, New York, 1997.

[20] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse. An empirical comparison between direct and indirect test result checking approaches. In *Proceedings of the Third International Workshop on Software Quality Assurance* (*SOQUA 2006*) (*in conjunction with the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*)), pages 6–13. ACM Press, New York, 2006.

[21] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD 2003*), pages 388–396. ACM Press, New York, 2003.

[22] D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21 (3): 24–35, 2001.

[23] D. P. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, San Francisco, California, 2003.

[24] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15 (2): 97–133, 2005.

[25] J. Mayer. On testing image processing applications with statistical methods. In *Software Engineering 2005* (*SE 2005*), Lecture Notes in Informatics, pages 69–78. Gesellschaft fu"r Informatik, Bonn, 2005.

[26] S. Melax. A simple, fast, and effective polygon reduction algorithm. *Game Developer Magazine*, pages 44–49, November 1998.

[27] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing?. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 164–173. IEEE Computer Society Press, Los Alamitos, California, 2003.

[28] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2000*), pages 30–39. ACM Press, New York, 2000.

[29] B. Meyer. *Eiffel: the Language*. Prentice Hall, New York, 1992.

[30] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 1998*), *ACM SIGSOFT Software Engineering Notes*, 23 (2): 82–92, 1998.

[31] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering* (*ICSE 2003*), pages 465–475. IEEE Computer Society Press, Los Alamitos, California, 2003.

[32] M. Segal and K. Akeley. *The OpenGL Graphics System: a Specification*. Version 2.0. Silicon Graphics, Mountain View, California, 2004.

[33] Y. Sun and E. L. Jones. Specification-driven automated testing of GUI-based Java programs. In *Proceedings of the 42nd Annual Southeast Regional Conference* (*ACM-SE 42*), pages 140–145. ACM Press, New York, 2004.

[34] M. Vanmali, M. Last, and A. Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17 (1): 45–62, 2002.