Techniques for Testing Scientific Programs Without an Oracle

Upulee Kanewala and James M. Bieman
Computer Science Department
Colorado State University, Fort Collins, CO, USA
Email: {upuleegk, bieman}@cs.colostate.edu

***Abstract*—The existence of an oracle is often assumed in software testing. But in many situations, especially for scientific programs, oracles do not exist or they are too hard to implement. This paper examines three techniques that are used to test programs without oracles: (1) Metamorphic testing, (2) Run-time Assertions and (3) Developing test oracles using machine learning. We examine these methods in terms of their (1) fault finding ability, (2) automation, and (3) required domain knowledge. Several case studies apply these three techniques to effectively test scientific programs that do not have oracles. Certain techniques have reported a better fault finding ability than the others when testing specific programs. Finally, there is potential to increase the level of automation of these techniques, thereby reducing the required level of domain knowledge. Techniques that can potentially be automated include (1) detection of likely metamorphic relations, (2) static analyses to eliminate spurious invariants and (3) structural analyses to develop machine learning generated oracles.**

***Index Terms*—Scientific software testing, Metamorphic testing, Metamorphic relation, Machine learning, Mutation analysis, Test oracles, Assertion checking**

I. Introduction

An *oracle* is used in software testing to check whether the program under test (PUT) produces the expected output when executed using a set of test cases. A program is considered *non-testable* [1] when an oracle does not exist or is impractical to implement even though theoretically possible. Weyuker identified three classes of programs that are non-testable [1]: (1) Programs written to find an answer that is previously unknown, (2) Programs producing so much output that it is practically impossible to verify all outputs and (3) Programs whose tester has a misconception about the program. Much scientific software falls into class 1 or 2 above.

In practice, especially when testing scientific software, test oracle implementation is ad-hoc. Even though developers of scientific software may be familiar with systematic testing, for example satisfying a test coverage criterion, systematic approaches are rarely used to develop test oracles. Often a domain expert looks at the results and determines that the results "look about right." Such an informal approach can miss subtle differences in the output such as failures caused by one-off errors. The systematic approaches that we describe here can be effective in improving the testing of scientific software.

The *precision* of an oracle is defined as its tolerance for errors [2]. An *imprecise* oracle will pass test cases that should have been failed or fail test cases that should have been passed. Approaches proposed in the literature for testing programs without oracles differ in their preciseness. A *null oracle* is the least precise oracle [2]. It is the oracle implicitly provided by the run-time system of the PUT and it can only detect if the PUT terminates abnormally or if it fails to terminate [2]. An *ideal oracle* provides pass/fail decisions without any mistake for all possible executions of the PUT with respect to its specification [2]. In practice, the most precise oracle one can develop is a *pseudo-oracle*. A *pseudo-oracle* is another program that is independently written using the same specification as the PUT [1].

In practice, the preciseness of oracles falls between a pseudo-oracle and a null oracle. In this work we examine three such techniques: (1) metamorphic testing, (2) run-time assertions, and (3) oracles built using machine learning techniques. Chen et al. [3] introduced *metamorphic testing* for testing programs without oracles. This technique operates by checking whether the program under test behaves according to a set of *metamorphic relations*, which specifies how a change to an input affects the output. The pass/fail judgment of test cases are determined using the set of metamorphic relations, and can be treated as an oracle.

In *run-time assertion checking*, assertions are embedded into the PUT. An assertion failure during the execution of a test case is reported as a program failure. Therefore the set of assertions act as the oracle.

Machine learning techniques can generate oracles for non-testable programs [4], [5]. This method works by building a classifier that predicts whether the output produced by the PUT for a test case is correct or not.

Since an oracle often does not exist for scientific programs [6], one (or a combination) of the above three techniques can be used. When employing these methods the test engineer has to be cautious about the *oracle properties* followed by the technique. In this paper, we review five studies [7], [8], [2], [5], [4] that employ these three techniques to test applications without oracles. We examine the properties of oracles produced by these techniques as well as their fault

This work is supported by Award Number 1R01GM096192 from the National Institute Of General Medical Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of General Medical Sciences or the National Institutes of Health.

finding ability. In addition we examine the level of automation that can be achieved and the amount of domain knowledge required when applying the methods.

The remainder of this paper is organized as follows: Section II describes the background on test oracles and machine learning. Metamorphic testing, its applications and open issues are discussed in Section III. Section IV discusses run-time assertion checking. Oracles created using machine learning techniques are described in Section V. Issues related to fault finding ability, automation and domain knowledge of the three techniques are discussed in Section VI. Section VII details the conclusions of this study.

II. Background

A. Test Oracles, their properties and comparison measures

A test oracle determines whether the output produced by a test case is correct according to the expected behavior of the PUT. The test engineer needs to be aware of the properties followed by the oracle used for testing in order to correctly evaluate the pass/fail judgments produced by the oracle.

Staats et al. define several general oracle properties [6]. They define the testing system to be a collection $(P, S, T, O, corr, corr_t)$ as follows:

- P is a set of programs.
- S is a set of specifications where $s \in S$ is an idealized specification of P.
- T is a set of tests.
- O is a set of oracles where $o \in O$ is a predicate such that $o \subseteq T \times P$. An oracle $o \in O$ determines for a given program and a test if the test passes. Following this definition we will use the term *false positive* to denote o signaling *true* when a fault is present in the code and, the term *false negative* to denote o signaling *false* when a fault is not present in the code.
- $corr$ is a predicate such that $corr \subseteq P \times S$. For a program $p \in P$ and a specification $s \in S$, $corr(p, s)$ implies p is correct with respect to s. Value of $corr(p, s)$ is generally unknown in practice.
- $corr_t$ is a predicate such that $corr_t \subseteq T \times P \times S$. It defines correctness with respect to a test $t \in T$. $corr_t(t, p, s)$ holds if and only if the specification s holds for program p when running test t.

1) Properties of Oracles:

Completeness: An oracle o is *complete* [6] with respect to a program p and specification s for a test case t if:

$$corr_t(t, p, s) \Rightarrow o(t, p)$$

For a test set T, the oracle o is said to be complete for p and s if:

$$\forall t \in T, corr_t(t, p, s) \Rightarrow o(t, p)$$

Completeness specifies that if the result obtained by running t on p is correct according to s, the oracle o will indicate that the test has passed. In general, oracles are designed to be complete, but there could be situations where this property fails. In particular, false negatives can be produced when using the techniques described in this paper. Use of an incomplete oracle can result in investing resources to fix a fault that is not actually present in the code.

Soundness: An oracle o is *sound* [6] with respect to a program p and a specification s for a test case t if:

$$o(t, p) \Rightarrow corr_t(t, p, s)$$

For a test set T, the oracle o is said to be sound for p and s if:

$$\forall t \in T, o(t, p) \Rightarrow corr_t(t, p, s)$$

Soundness specifies that if o indicates that t has passed, then p is correct with respect to t according to s. Oracles used in practice are rarely sound since a fault can be manifested in variables or states that are not observed by o.

2) Oracle Comparison Measures: Staats et al. [6] recently proposed two measures for systematically comparing oracles: (1) *power comparison* and (2) *probabilistic comparison*. These measures compare the relative usefulness of oracles based on their ability to find faults. Both power and probabilistic comparison measures are defined for complete oracles, and they do not address the effect of false negatives.

Power Comparison: For a program p and specification s, an oracle o_1 has a power greater than or equal to oracle o_2 with respect to a test set T (written as $o_1 \geq_T o_2$) if:

$$\forall t \in T, o_1(t, p) \Rightarrow o_2(t, p)$$

That is if o_1 fails to detect a fault for some test then o_2 will not detect that fault either.

Here, the power of oracles are compared relative to a specific test set. Therefore power relationship between oracles may change with different test sets. If for all possible test sets o_1 has power greater than or equal to o_2, then o_1 is said to have a power *universally* greater than o_2.

To satisfy $o_1 \geq_T o_2$, o_1 has to detect all the faults detected by o_2. But oracles created using different techniques (e.g. metamorphic testing vs. assertion checking) would usually detect different faults. Therefore power relation will not be useful when comparing such oracles [6].

Probabilistic Comparison (PB): To facilitate a comparison between oracles created using different methods Staats et al. [6] developed the PB relation. An oracle o_1 is said to be *PROBBETTER* (*PB*) than oracle o_2 (written as $o_1\, PB_T\, o_2$) with respect to a test set T for a program p, if o_1 is more likely to detect a fault than o_2 for a randomly selected $t \in T$. This relation can be used to compare oracles with respect to a specific test set.

B. Machine Learning

Machine Learning methods focus on providing the ability for computer programs to make better decisions based on experience [9]. Usually, the set of examples used by a machine learning algorithm are divided into two subsets: a *training set* and a *test set*. The *training set* is used to create the predictive model (training/learning), while the *test set* is used to evaluate the performance of the predictive model (testing). *Supervised learning* is one machine learning method, which employs a set of labeled examples to learn a target function. The target function maps the input to a desired set of outputs (labels). A predictive model that assigns a *class* (label) to an example is called a *classification* model. Input to a supervised classification algorithm is a set of training data $S = \{s_1, s_2, ..., s_n\}$. Each vector $s_i = x_1, x_2, ..., x_m, c_i \in S$ is called a *training instance/example*, where x_j is a *feature* and c_i is the class label of the training instance s_i. In *binary* classification the class label can take only one of two possible values (i.e. *positive examples* and *negative examples*).

1) Decision trees (DT): In decision tree learning, the target function is a decision tree in which internal nodes test a feature in the input and leaf nodes assign a label. A trained decision tree model can predict (assign) a label for an previously unseen example. J48 is the Java implementation of the C4.5 [10] decision tree generation algorithm, from the WEKA [11] tool kit. When choosing a feature for an internal node, C4.5 chooses the feature with the highest information gain [12].

2) Cross validation: The *k-fold cross-validation* technique evaluates how a predictive model would perform on previously unseen data. In *k-fold cross-validation* the data set is randomly partitioned into k subsets. Then $k - 1$ subsets are used for training and the remaining subset is used for testing. This process is repeated *k* times in which each of the *k* subsets is used to evaluate the performance. In *stratified k-fold cross-validation*, *k* folds are partitioned in such a way that the folds contain approximately the same proportion of positive and negative examples as in the original data set.

3) Performance measures: **Accuracy** is the percentage of correct predictions made by the predictive model.

$$Accuracy = \frac{\textit{true positives} + \textit{true negatives}}{\textit{true positives} + \textit{true negatives} + \textit{false positives} + \textit{false negatives}}$$

Area under the receiver operating characteristic curve (AUC) is a performance measure that is used to evaluate the predictive models. It is a measure of the quality of rankings given by a model and is widely used to compare the performance of predictive models in machine learning. AUC measures the probability that a randomly chosen negative example will have a smaller estimated probability of belonging to the positive class than a randomly chosen positive example [13]. A higher AUC value indicates that the model has a higher predictive ability. AUC takes a value in the range $[0, 1]$. A classifier with $AUC = 1$ is considered a perfect classifier while, a classifier with $AUC = 0.5$ is considered as a classifier that makes random predictions.

We now examine the three techniques that deal with the oracle problem. We examine these techniques in terms of oracle properties, fault finding measures, potential automation and required domain knowledge.

```
public static int addValues(int a[]){
      int sum=0;
      for(int i=0;i<a.length;i++){
            sum+=a[i];}
      return sum;}
```

Fig. 1. Function for calculating the sum of elements in an array

III. Metamorphic Testing (MT)

Metamorphic testing tests a PUT by checking whether the PUT behaves according to a set of metamorphic relations (MRs). A *metamorphic relation* specifies how a particular change to the input of the program would change the output. For a function *f*, a metamorphic relation *R* would express a relationship among multiple inputs x_1, x_2,...,x_n for $n > 1$ and their corresponding output values $f(x_1)$, $f(x_2)$,...,$f(x_n)$ [3]. Since MRs specify a relationship among multiple executions, they can be used for testing even if the correct output of individual executions are unknown.

Violation of a metamorphic relation occurs when a change in the output differs from what is predicted by the considered metamorphic relation. Satisfying a metamorphic relation does not guarantee that the program is implemented correctly. However, a violation of a metamorphic relation indicates that the program contains faults.

A. MT Process

1) Identifying metamorphic relations: The most important step in MT is identifying a set of MRs [7]. MRs are derived using the domain knowledge of the PUT. Two approaches can be used to develop MRs: (1) MRs that are enumerated specifically for the PUT using its specification, and (2) MRs enumerated from the general expectation of the user about the PUT. MRs created using method 1 will be *necessary* properties of the PUT. Therefore a violation of such a MR indicates a fault in the PUT, and thus can be used for verification. Method 2 will not always result in a necessary property of the program. They will be *desired* properties of the PUT and they can be utilized for validation [7]. These MRs will check whether the PUT satisfies the expected requirements of the user.

Consider the function in Figure 1 that calculates the sum of integers in an array *a*. The following two properties can be treated as MRs for testing this function:

MR$_1$: Randomly permuting the order of the elements in a should not change the result

MR$_2$: Adding a positive integer k to every element in a should increase the result by $k \times length(a)$

Not all MRs have the same fault detection ability. MRs that enforce an equality relationship between the initial and follow-up test cases are preferred over MRs that enforces a non-equality relationship, since an equality relationship can be violated more easily than a non-equality relationship [7].

2) Creating test cases: MT requires creating *initial* and *follow-up* test case pairs, according to the input changes required by the identified MRs. Initial test cases can be generated using testing techniques such as random testing, fault based testing, etc. Then, follow-up test cases are created by modifying the initial test cases according to the modifications required by the MRs. Chen et al. found that MRs that produce different execution traces for the initial and follow-up test cases will be more effective than those with similar execution traces for both test cases [14].

3) Failure detection: Each of the initial and follow-up test case pairs are executed to check whether the output change complies with the change predicted by the corresponding MR. A run-time violation of a metamorphic relation indicates a fault or faults in the PUT. Since metamorphic testing checks the relationship between inputs and outputs of multiple executions of the PUT, this method can be used when the result of individual executions are not known.

B. Empirical Studies

MT is used for testing programs that do not have oracles in different domains. MT was shown to be effective in bioinformatics [15], health care simulations [16], Monte Carlo modeling [17], computer graphics [18] and for testing programs with partial differential equations [19].

1) Applying MT to test machine learning classifiers: Xie et al. applied MT for testing machine learning classifiers [7]. They used MT for testing and validating *k-nearest neighbor* (*k-NN*) and *Naïve Bayes classifier* (*NBC*). These programs are non-testable programs because they use a complicated logical and computational process [7].

Xie et al. developed 11 MRs using the general anticipated behavior of supervised machine learning classifiers. They conducted two experiments in this work:

(1) Testing the k-NN and NBC implementations in the Weka [11].
(2) A mutation analysis on the same algorithms.

Results of the first experiment showed that although k-NN did not violate any of the MRs that were necessary properties, NBC violated several MRs that were necessary properties. These violations indicate faults in the NBC implementation in Weka. There were some violations of MRs used for validation in both programs. Although these violations might not indicate an actual fault in the program, they indicate that there could be deviations from the expected behavior.

In the second experiment, a mutation analysis was conducted to empirically measure the effectiveness of MT. Xie et al. created mutants using the traditional mutation operators in MuJava and randomly selected 30 mutants for each of the algorithms. For testing k-NN, 11 MRs were used while nine MRs were used for testing NBC with 300 randomly generated initial test cases. Xie et al. reported that 90.5% mutants were killed in k-NN and 90.9% mutants were killed in NBC.

Finally, Xie et al. compared the effectiveness of MT with cross validation analysis. The experiment used three simulated data sets. A mutant is considered as "killed" if the error-rate obtained by cross validation is significantly different from the expected results for the simulated dataset. Results showed that 31.6% mutants survived in k-NN and 40% of mutants survived in NBC. Figure 2 shows the percentage of mutants killed by MT and cross-validation.

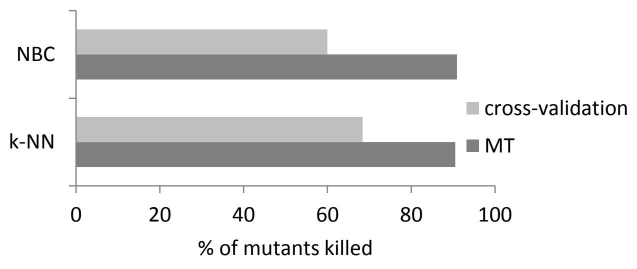


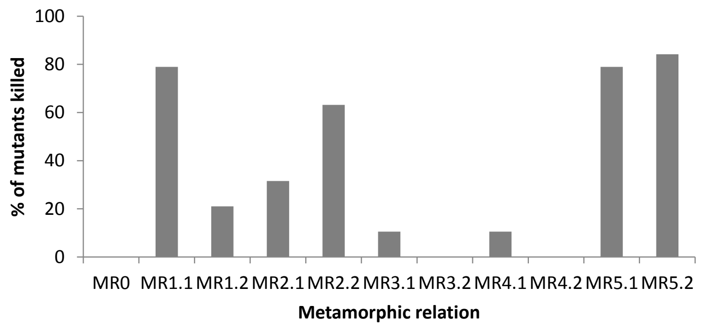
Fig. 2. Effectiveness of MT and cross-validation [7]

Xie et al. [7] applied MT for testing complex real world applications. They used general anticipated properties of the domain of the PUT when developing MRs, which is easier than specifically developing MRs for the PUT. But, this approach can produce MRs that are not necessary properties of the PUT. Xie et al. [7] manually proved that some of these properties are necessary properties of the PUT. Then they only used these necessary properties for the mutation analysis thus making the oracle used in verification a complete oracle. Manually proving that certain properties are necessary properties of the PUT requires a good understanding of the algorithms that are used by the PUT.

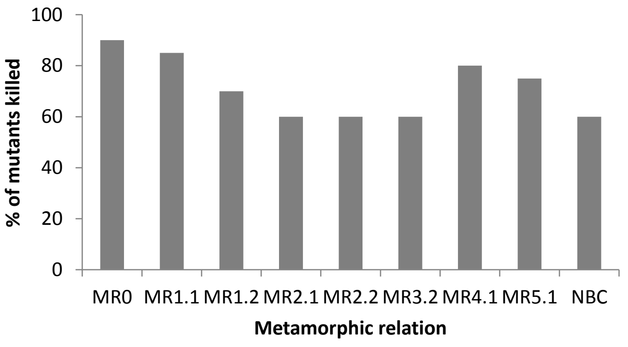
As with most of the oracles used in practice this is not a sound oracle, since an error can be manifested in a state or a variable that cannot be uncovered by the set of MRs used here. Xie et al. claim that MT could be more effective in detecting faults than a traditional method (cross validation) used for testing in the domain. But Xie et al. have used different test cases when applying the two methods, making it difficult to fairly compare the two methods. A fair comparison between these two methods can be performed by using the PB measure in Section II-A2, since it compares the effectiveness for a specific test set.

Figures 3a and 3b show the percentage of mutants killed by each MR in k-NN and NBC programs respectively. Figure 3a shows that three MRs could not kill any mutants. Further, several MRs that performed poorly in k-NN performed significantly better in NBC. In addition several mutants could not be killed even though the modified statements were executed. The k-NN implementation was executed $3600(300+300\times11)$ times and the NBC implementation for $3000(300+300\times9)$ times and detected 19 and 20 mutants respectively. Such a large number of executions is not practical when testing applications with long execution times.

Further, randomly selected 30 mutants from complex algorithms might not be representative of faults made by a typical programmer. In addition the mutants only cover 6 mutation operators, while there are 12 mutation operators supported by MuJava. Therefore the mutants that Xie et al. used in the experiment may not adequately represent the faults made by a programmer.



(a) Percentage of mutants killed by each MR for k-NN



(b) Percentage of mutants killed by each MR for NBC

Fig. 3. Percentage of mutants killed by each MR [7]

2) Applying MT for unit and integration testing: Just et al. conducted an empirical study to evaluate the effectiveness of MT for integration testing [8]. They used MT to test an implementation of the jpeg2000 encoder, which is a modular system written in Java. This application faces the oracle problem when randomly generated inputs are used for testing [8].

Just et al. used five MRs for testing individual modules as well as the whole application. They used both traditional and class-based mutants generated by MuJava. Class-based mutants represent faults introduced during integration. Since the faulty statement has to be executed in order to be detected by any oracle, Just et al. first assessed the initial inputs that were randomly generated. They used 1977 traditional mutants and 206 class-based mutants in the experiment. After selecting the initial inputs, follow-up test cases were created according to the input modifications required by the MRs.

Just et al. calculated the number of mutants killed by each MR in each module and in the complete application. Their results show that the effectiveness of individual MRs ranges from $65\% - 97\%$ between subsytems for traditional mutants. In addition, results show that individual MRs are less effective in detecting class-based mutants (killed $41\% - 73\%$) when compared with traditional mutants (killed $81\% - 91\%$). Results also show that MRs that reported the highest mutation detection scores when testing individual units (i.e. traditional mutants) did not achieve the highest mutation scores when testing the whole system (i.e. for class-based mutants).

Further, Just et al. calculated the mutation detection score collectively achieved by the MRs. Their results show that the mutation detection score can be improved for both traditional and class-based mutants by applying several MRs as combinations. In addition, they conclude that by testing with only the most effective MRs, one can improve effectiveness while reducing the time spent on testing, rather than testing with all the MRs. Finally, they combined the two MRs that reported the highest effectiveness for traditional and class-based mutants to build a single complex MR. Using this complex MR, they were able to significantly improve the percentage of class-based mutants that were killed up to 96.6%.

Just et al. found that using complex MRs, which were developed by combining MRs used for unit testing, the effectiveness of integration testing can be improved. They selected test cases that achieve a high mutation score, in order to limit the affect of inadequate coverage. Five MRs were developed using necessary properties of the PUT producing a complete but not sound oracle for testing. Just et al. do not demonstrate that class-based mutants represent actual integration errors made by software developers.

C. Limitations, Unsolved Problems and Future Work on MT

Automatically detecting likely MRs for a program: Currently the test engineer or the programmer has to identify the MRs manually, which could result in missing important MRs. The development of automated methods for detecting likely MRs for the PUT is an open problem.

Identifying effective MRs: The experiments described in Sections III-B1 and III-B2 found that the fault finding ability of MRs varies significantly. Developing methods to identity the set of MRs that are most effective at detecting faults for a particular program is an open problem.

Effectiveness of complex MRs compared to simple MRs: Fault finding effectiveness of complex MRs developed by combining two or more MRs is another open problem. By combining several MRs to form a single complex MR, the number of executions required in testing can potentially be reduced.

Identifying limitations of MT: One of the challenges faced when using MT is to determine whether the PUT is adequately tested. Even if 100% of the code is covered by the test cases, a pair of incorrect outputs might still satisfy a MR used for testing. The experiments described in Sections III-B1 and III-B2 found several mutants that could not be killed using MT. Identifying whether these mutants could be killed with much stronger MRs or whether there are certain mutants that could never be killed using MT is an open problem.

IV. Assertion Checking

An *assertion* is a boolean expression or a constraint that is used to verify some necessary property of the PUT [20]. Usually, assertions are embedded into the source code of the PUT and evaluated when a test case is executed. Widely used programming languages such as C and Java provide built-in support for assertions.

In the context of testing non-testable programs, assertions can be used to check some degree of correctness of the PUT [21]. For example assertions can be used to verify whether the output is within an expected range (even though the actual value is unknown) or some known relationships between program variables are maintained, etc. One advantage of assertions is that they can be used to detect faults that are executed with the test case but not resulted in failures (i.e. that do not propagate to the output).

A. *Identifying Assertions*

Coppit et al. developed a method where they first develop a formal specification of the PUT and then convert the specification into assertions [22]. This method might be not practical in the context of most scientific programs, since they are exploratory in nature and their specifications can face frequent changes [23].

Invariant detection tools like Daikon [24] can assist the creation of assertions. Daikon executes the program several times and reports the properties that were true over these executions. These assertions are applied in subsequent program runs. One limitation of Daikon is that the invariants are limited to function pre- and post-conditions. Therefore Daikon cannot be used to create assertions that are inside the body of a function [21].

These tools can generate incorrect invariants, even when using large test suites [25]. Therefore human intervention is required to select correct invariants before using them as test oracles. However a study conducted by Staats et al. showed that humans also fail to correctly classify automatically generated invariants [26].

B. *Empirical Studies*

1) Effectiveness of assertions as oracles: Shrestha et al. conducted an empirical study to evaluate the effectiveness of using assertions as oracles [2]. They compared assertion checking and the null oracle in terms of the fault detection capability. This experiment used 15 Java classes as test subjects; 7 classes were obtained from JDK 1.4.2 and 8 classes from two sample programs. The authors used a set of JML (Java Modeling Language) assertions provided by a third party (JML project[1]) for these classes. Test cases were created using the branch coverage criterion (targeting 90%). JUnit[2] test cases were written to execute the programs with the input values.

The effectiveness of assertions is measured in terms of the percentage of the mutants that were killed. Mutants of the subject programs were created using the traditional mutation operators provided by the MuJava mutation engine. To evaluate the effectiveness of the null oracle, Shrestha et al. executed all the test suites against all the mutants and recorded the pass/fail information. Then the same test suites were executed while the JML assertions in the programs were enabled. Mutants that were not killed by any of the test cases were

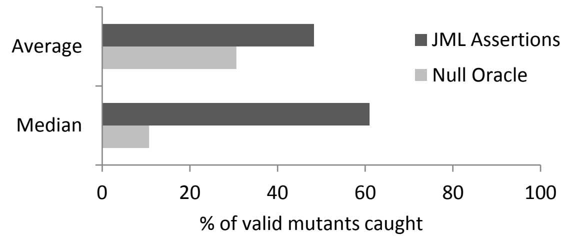


Fig. 4. Average and median effectiveness of the null oracle and JML assertions [2]

manually inspected to check whether they were equivalent mutants. Results were recorded for 104 methods for the null oracle and for 56 methods for JML annotations.

As shown in Figure 4, on a per method basis the median effectiveness of null oracle is only 10.7%. For assertion checking the median percentage of faults caught was 61% (after removing the mutants caught by the null oracle). By analyzing the distribution of effectiveness of detecting faults of the null oracle and assertions on a per method basis, Shrestha et al. reported that the null oracle is not effective in catching bugs beyond 50%. They also observed that $95\% - 100\%$ of the faults were caught in 39% of the methods and less than 5% of the faults were caught in 46% of methods using assertions. Very few methods have between 5% and 95% of the faults caught. The conclusion is that assertion checking is either very effective or totally ineffective depending on the method.

C. *Limitations, Unsolved problems and Future Work in Assertion Checking*

Minimize spurious invariants detected by automatic invariant detection methods: As mentioned earlier, creating assertions using an automatic invariant detection tool may be more effective than creating specification based assertions when testing scientific programs. Further investigations can evaluate the effectiveness of tools such as Daikon, especially when testing non-testable scientific software. These tools can detect spurious invariants producing incomplete oracles that may signal faults that are not actually present in the code. Therefore, we need methods to reduce the number of incorrect invariants generated by such tools, such as using static analysis techniques that can falsify incorrect assertions [26].

Investigate utility of assertions when testing programs without oracles: Few studies used assertion checking for testing non-testable programs [21], [27]. Additional empirical studies should be conducted using assertions as oracles targeting non-testable scientific programs.

V. Machine Learning Based Methods for Developing Oracles

Machine learning techniques have been used for developing oracles for non-testable programs [5], [4]. Both black-box (features developed using inputs and outputs of the program) [5], [4] and white-box (features developed using the internal structure of the program) [28] features have been used to train the classifiers used as the oracle. Using white-box

[1]http://www.eecs.ucf.edu/∼leavens/JML//index.shtml

[2]http://junit.sourceforge.net/

features (such as execution traces) for developing classifiers will be less reliable than using black-box features, since these features may suffer from the effects of coincidentally correct test execution data [5].

A. Empirical Studies

1) Developing an oracle for mesh simplification programs [5]: Chan et al. developed an oracle for testing a *mesh simplification* algorithm using machine learning [5]. These applications face the oracle problem since neither automatic pixel-by-pixel comparison nor manual comparison will be reliable [5]. Chan et al. developed a classifier that can classify the output produced by the PUT as *passed* or *failed*. This classifier was trained using a set of black-box features extracted from outputs generated from an existing program called a *reference model*, which implements the same functionality as the PUT.

Figure 5 shows an overview of the proposed method. Let P be the mesh simplification program under test (PUT) and R be a reference model for P. Let $M = \{m_1, m_2, ..., m_k\}$ be a set of 3D polygonal models that can be used as inputs to P and R. First, a set of faulty versions of R were generated using a mutation generation engine. Then the set of inputs were executed on the original R as well as its faulty versions. Then a set of features were extracted from the produced outputs. Features obtained from R were labeled as *passed* and the features obtained from mutants were labeled as *failed*. Finally, these features were used to train a classifier C. When testing the PUT, the same set of features were extracted from the PUT by executing test cases. Then these feature vectors were provided to C. The classifier C labeled the feature vector as *passed* or *failed*. A test case that produced a *passed* feature vector was considered as a test case that reveals no failure, while a test case that produced a *failed* feature vector was considered as a test case that reveals a failure.

In the experiment, Chan et al. used a set of classification features based on the strength of image frequencies and light effects. They tested the proposed method using four image segmentation implementations that were written in Java. They used 44 open-source 3D polygonal models as their test cases. The authors selected the C4.5 classification algorithm to create the classifier based on the results of a pilot study.

The evaluation is in terms of accuracy (percentage of test cases that were correctly classified), effectiveness (percentage of failed test cases that were properly classified) and robustness (percentage of passed test cases that were properly classified). Based on these observations Chan et al. conclude:

(1) An accurate, effective and robust test oracle can be developed using a resembling reference model.
(2) If a resembling model does not exist, a less sophisticated model should be used. Classifiers developed using simple reference models achieve a higher accuracy while providing a low effectiveness.
(3) Dissimilar reference models results in less robust classifiers.

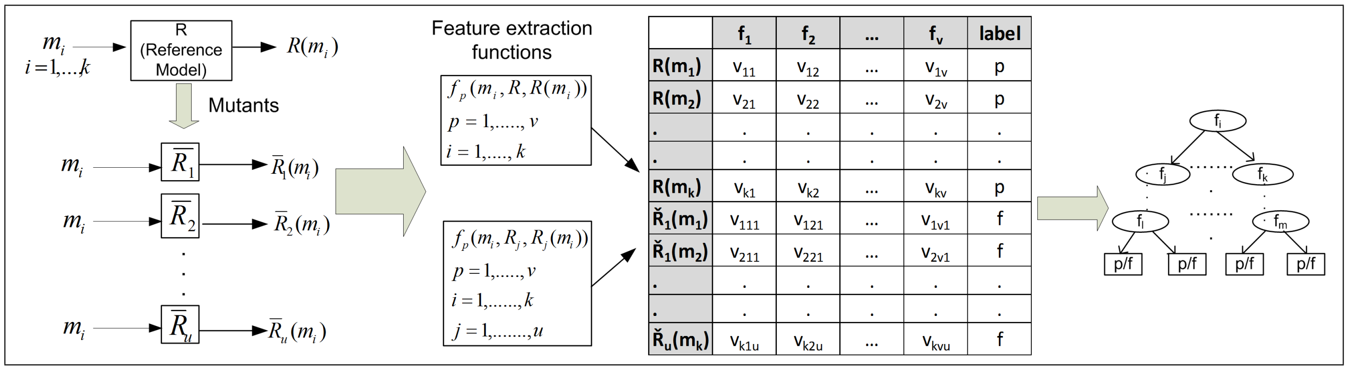
The resulting classifier can produce pass/fail judgment of test cases based on the knowledge obtained from a reference model. This oracle is incomplete, i.e. it can produce false negatives. It is also not a sound oracle since an error can occur in an unseen example during the learning phase. Feature selection plays an important role in the effectiveness of this method. Therefore a domain expert needs to be involved in selecting the set of features for developing the classifier. After selecting an appropriate set of features, the oracle creation can be done automatically.

In this work Chan et al. used outputs created from reference models for learning the classifier. Therefore this method assumes that independently developed reference models would not result in same faults. But this independent assumption has been contradicted empirically by Brilliant et al. [29]. Thus Chan et al.'s method can suffer from similar faults present in the reference models.

2) Developing an oracle for a image segmentation program: Frounchi et al. [4] developed an oracle for verification and validation of an image segmentation algorithm using machine learning. They developed a machine learning classifier that determines whether a segmentation created by a subsequent version of the program is consistent with a previously verified image segmentation of the same original image.

During the training phase, a pair of segmentations were obtained from version v_x and $v_y (y < x)$ of the segmentation program for each input. Then a set of predetermined similarity measures were obtained for these segmentation pairs. Using a domain expert, these pairs of segmentations were labeled as *consistent* (if both segmentations are correct) or *inconsistent*(if one of the segmentations is correct and the other one is incorrect). This manual labeling has to be done for at least the segmentation pairs created by the first two versions of the algorithm. Using the similarity measures and the label given for each image segmentation pair i, a tuple of the form $(sm_{i1}, ..., sm_{ik}, consistency)$ were created. Value sm_{ij} denotes the similarity measure j obtained for image pair i. Class label $consistency$ can take two values $consistent$ or $inconsistent$. This set of tuples was used as the training set to build the classifier. After the classifier achieves a satisfactory accuracy, it can be used to verify whether a segmentation produced by a subsequent version of the program is consistent with previously verified segmentations.

Frounchi et al. evaluated the performance of the learned classifier using 10-fold-cross validation. They used a cardiac left ventricle segmentation algorithm as the PUT. The test set consist of 181 CT-Scan images. Three classification algorithms (J48, JRIP and PART) were used in the experiment. Accuracy and AUC are used as performance measures. The three classification algorithms used in the experiment gave equivalent performance indicating that the choice of the machine learning algorithm is not significant. The results show that using a wrapper to pre-select attributes can improve the classifier performance. The decision tree classifier created with J48 using all the 18 similarity measures and a wrapper for selecting suitable measures reported the best performance. It achieved an average accuracy of 95% and an AUC measure of 0.95.



	f_1	f_2	...	f_v	label
$R(m_1)$	v_{11}	v_{12}	...	v_{1v}	p
$R(m_2)$	v_{21}	v_{22}	...	v_{2v}	p
.	.	.	.	.	.
.	.	.	.	.	.
$R(m_k)$	v_{k1}	v_{k2}	...	v_{kv}	p
$\check{R}_1(m_1)$	v_{111}	v_{121}	...	v_{1v1}	f
$\check{R}_1(m_2)$	v_{211}	v_{221}	...	v_{2v1}	f
.	.	.	.	.	.
.	.	.	.	.	.
$\check{R}_u(m_k)$	v_{k1u}	v_{k2u}	...	v_{kvu}	f

Fig. 5. Overview of the proposed method by Chan et al. [5]

In summary, the classifier can automatically determine the correctness of image segmentations using machine learning. Since the classifier achieved an accuracy less than 100%, the oracle developed using this method is incomplete. The oracle is also not sound since it will only have the knowledge obtained from a set of finite examples. This method requires selecting a set of similarity measures that will help to determine the consistency of image segmentation pairs. This requires an adequate understanding of the domain.

For the image segmentation algorithm used in this experiment, none of the versions produced incorrect segmentation for both versions v_x and v_y $(y < x)$. A different result may occur when developing an algorithm from scratch. Most of the segmentations produced by initial versions might be incorrect making them unsuitable for training the classifier. Therefore, several iterations with manual verification might be required before achieving an effective classifier. So the optimal number of iteration required for achieving an effective classifier needs to be further investigated.

B. Limitations, Unsolved Problems and Future Work in ML Based Methods

Minimizing the effects of coincidental correctness in structural (white-box) features: Both of the applications discussed in Sections V-A1 and V-A2 are non-testable programs since they produce complex outputs that are difficult to verify manually. Both of these applications are in the domain of image processing, where there are standard measures to measure the similarities/differences between outputs. Therefore, applying this method in other domains (without such measures) will require an in-depth investigation on how to measure the similarity between the outputs. On the other hand it might be more effective to investigate the effectiveness of using classifiers developed using white-box features in such situations while mitigating the effects of coincidentally correct executions.

VI. Discussion

A. Fault Finding Effectiveness

Performing a complete comparison between the three techniques in terms of their fault finding ability is not possible due to the vast differences in the programs the techniques have been applied to. But it is possible to perform the comparison for a fixed program. Two studies compare the fault finding ability of MT and assertion checking for a fixed set of programs.

Zhang et al. [27] compared testing effectiveness of MT and assertion checking in terms of mutation detection ratio using graduate student subjects. They found that MT is more effective than assertion checking in finding faults. Murphy et al. [21] compared MT and assertion checking when testing machine learning and non-deterministic applications. Murphy et al. used the Daikon invariant detection tool [24] for automatically detecting invariants. The MRs for MT were developed manually. The results show that MT is more effective in detecting faults than assertion checking.

An important factor that determines the fault finding ability is the test set. Much research assume the presence of an ideal oracle when comparing coverage criteria [6]. Therefore, conclusions in these studies might not apply to non-testable programs. For example, consider a situation where we use a pseudo-oracle (which only considers the output) and assertion checking (that considers the internal variables) with test cases developed using statement and branch coverage criteria. Assume that the PUT contains a fault f that can be caught using any test set satisfying the statement coverage but that does not propagate to the output. Even though in theory f should be revealed by a test set satisfying the branch coverage criteria (since branch coverage subsumes statement coverage), if the test suite satisfying the branch coverage is paired with the pseudo-oracle it might not reveal the fault while the test suite satisfying the statement coverage paired with assertion checking would be guaranteed to uncover the fault [6]. Therefore the test engineer should not assume that more faults can be detected by adding more test cases to satisfy a certain coverage criteria. Further it might be more effective to derive more MRs (when using MT) or more assertions (when using assertion checking) to improve the fault finding ability than adding more test cases to satisfy a certain coverage criteria.

B. Automation

The level of automation achieved by the three methods discussed in this paper differs. Tools such as Daikon can be used for developing assertions [24]. But such tools based on dynamic approaches can generate spurious invariants if there are not enough executions of the PUT.

With MT, follow-up test case generation, test execution and failure detection steps have been automated when the set of MRs satisfied by the PUT is provided [30]. But there have been no previous attempts to develop methods to automatically detect likely MRs. We developed a technique based on machine learning to detect these properties [31]. In addition it might be possible to use a dynamic approach similar to Daikon [24] for automatically detecting likely MRs.

Both of these automation approaches could add spurious assertions/MRs to these methods, producing incomplete oracles. Therefore when evaluating these techniques (MT with automatically detected MRs and assertion checking with automatically detected invariants) one has to consider the effects of false negatives. With false negatives, the power or probabilistic comparison measures cannot be applied directly. New comparison measures should be developed that account for the effects of falsely signaled faults.

Automation methods should also focus on minimizing the number of spurious assertions/MRs as much as possible since the test engineer would have to spend her time trying to find out whether the signaled failure is due an actual fault in the code.

C. Domain knowledge

The reviewed methods require different levels of domain knowledge. For MT, depending on the MRs used for testing, one has to be familiar with general properties of the domain or specific properties for the implementation. Assertion checking is similar. By developing automated methods to detect assertions and MRs with high accuracies, the domain knowledge required by the test engineer can be reduced significantly.

When developing oracles using machine learning methods, one needs to be familiar with the machine learning techniques in addition to the domain knowledge of the PUT. When using black-box features to develop the classifiers, similarity measures used as features will determine the accuracy of the classifier. Therefore, deep understanding of these measures is required to develop effective oracles. In addition having familiarity with different machine learning algorithms, feature selection methods and performance measures is important to successfully develop oracles using ML techniques.

VII. Conclusions

Scientific programs often lack oracles because they are written to find out a previously unknown answer or because they produce complex outputs. As a result, ad-hoc methods are widely used to test scientific software. In this paper we examined three techniques that can be used to systematically test programs without oracles: metamorphic testing, assertion checking and oracles created using machine learning methods. These techniques have been used to test scientific programs in different areas such as machine learning and image processing. Oracles produced by these techniques may not satisfy the oracle properties satisfied by traditional oracles. for example oracles created using machine learning methods produce incomplete oracles. Therefore the test engineer needs to be cautious when applying these techniques.

Developing techniques to automatically detect metamorphic relations would improve the automation of metamorphic testing. In addition, spurious invariants generated by automatic invariant detection tools have to be minimized in order to use them effectively for testing scientific programs. Investigating the possibility of using white-box features for developing machine learning based oracles would help to apply these oracles to a wider range of programs. Since these automation approaches could generate incomplete oracles, falsely signaled faults have to be considered when comparing the effectiveness of these oracles.

References

[1] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[2] K. Shrestha and M. Rutherford, "An empirical evaluation of assertions as oracles," in *Software Testing, Verification and Validation (ICST), 2011 IEEE 4th Int. Conf. on*, 2011, pp. 110 –119.

[3] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, 2003.

[4] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan, "Automating image segmentation verification and validation by learning test oracles," *Inf. Softw. Technol.*, vol. 53, no. 12, pp. 1337–1348, Dec. 2011.

[5] 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," *Journal of Systems and Software*, 2008.

[6] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 391–400.

[7] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544 – 558, 2011.

[8] R. Just and F. Schweiggert, "Automating unit and integration testing with partial oracles," *Software Quality Journal*, vol. 19, pp. 753–769, 2011.

[9] D. Zhang and J. J. Tsai, *Advances in Machine Learning Applications in Software engineering*. Idea Group Publishing, 2007.

[10] J. R. Quinlan, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[12] H. Zhu, "On information and sufficiency," *Annals of Statistics*, 1997.

[13] J. Huang and C. Ling, "Using AUC and accuracy in evaluating learning algorithms," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 3, pp. 299 – 310, march 2005.

[14] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou, "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 (JIISIC 2004)*, 2004, pp. 569–583.

[15] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing." *BMC Bioinformatics*, vol. 10, 2009.

[16] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil, "On effective testing of health care simulation software," in *Proceedings of the 3rd Workshop on Software Engineering in Health Care*, ser. SEHC '11. New York, NY, USA: ACM, 2011, pp. 40–47.

[17] J. Ding, T. Wu, D. Wu, J. Q. Lu, and X.-H. Hu, "Metamorphic testing of a Monte Carlo modeling program," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 1–7.

[18] R. Guderlei and J. Mayer, "Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *Quality Software, 2007. QSIC '07. Seventh International Conference on*, October 2007, pp. 404 –409.

[19] T. Y. Chen, J. Feng, and T. H. Tse, "Metamorphic testing of programs on partial differential equations: A case study," in *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, ser. COMPSAC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 327–333.

[20] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 25–37, May 2006.

[21] C. Murphy and G. E. Kaiser, "Empirical evaluation of approaches to testing applications without test oracles," Dep. of Computer Science, Columbia University, Tech. Rep. CUCS-039-09, 2010.

[22] D. Coppit and J. Haddox-Schatz, "On the use of specification-based assertions as test oracles," in *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, april 2005, pp. 305 –314.

[23] K. S. Ackroyd, S. H. Kinder, G. R. Mant, M. C. Miller, C. A. Ramsdale, and P. C. Stephenson, "Scientific software development at a research facility," *IEEE Softw.*, vol. 25, no. 4, pp. 44–51, Jul. 2008.

[24] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 213–224.

[25] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 93–104.

[26] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding user understanding: determining correctness of generated program invariants," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 188–198.

[27] Z. Zhang, W. K. Chan, T. H. Tse, and P. Hu, "Experimental study to compare the use of metamorphic testing and assertion checking," *Journal of Software*, vol. 20, no. 10, pp. 2637–2654, 2009.

[28] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 195–205, Jul. 2004.

[29] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an n-version software experiment," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 238–247, Feb. 1990.

[30] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 436–445.

[31] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," 2013, manuscript submitted for publication.