Preprint: Proc. IEEE Int. Symposium on Software Reliability Engineering (ISSRE), pp. 1-10, 2013.

Using Machine Learning Techniques to Detect Metamorphic Relations for Programs without Test Oracles

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

***Abstract*—Much software lacks test oracles, which limits automated testing. *Metamorphic testing* is one proposed method for automating the testing process for programs without test oracles. Unfortunately, finding the appropriate *metamorphic relations* required for use in metamorphic testing remains a labor intensive task, which is generally performed by a domain expert or a programmer. In this work we present a novel approach for automatically predicting metamorphic relations using machine learning techniques. Our approach uses a set of features developed using the control flow graph of a function for predicting likely metamorphic relations. We show the effectiveness of our method using a set of real world functions often used in scientific applications.**

***Index Terms*—Software testing, Metamorphic testing, Metamorphic relation, Machine learning, Mutation analysis, Scientific software testing, Test oracles, Decision trees, Support vector machines**

I. Introduction

One of the greatest challenges in software testing is the oracle problem. Automated testing requires automated test oracles, but such oracles may not exist. This problem commonly arises when testing scientific software. Many scientific applications fall into the category of "non-testable programs" [1] where an oracle is unavailable or too difficult to implement. In such situations, a domain expert must check that the output produced from the application is correct for a selected set of inputs. Further, Sanders et al. [2] found that, due to a lack of background knowledge in software engineering, scientists conduct testing in an unsystematic way. This situation makes it difficult for testing to detect subtle errors such as one-off errors, and hinders the automation of the testing process. A recent survey conducted by Joppa et al. [3] showed that when adopting scientific software, only 8% of the scientists independently validate the software and the others choose to use the software simply becuse it was published in a peer-reviewed journal or based on personal opinions and recommendations. Therefore undetected subtle errors can affect findings of multiple studies that use the same scientific software. Techniques that can make it easier to test software without oracles are clearly needed.

This project 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.

Metamorphic testing is a technique introduced by Chen et al. [4] that can be used to test programs that do not have oracles. This technique operates by checking whether the program under test behaves according to an expected set of properties known as *metamorphic relations*. A *metamorphic relation* specifies how a particular change to the input of the program would change the output [5]. Violation of a metamorphic relation occurs when the change in the output differs from what is specified by the considered metamorphic relation. Satisfying a particular metamorphic relation does not guarantee that the program is implemented correctly. However, a violation of a metamorphic relation indicates that the program contains faults. Previous studies show that metamorphic testing can be an effective way to test programs without oracles [5], [6]. Enumerating a set of metamorphic relations that should be satisfied by a program is a critical initial task in applying metamorphic testing [7], [8]. Currently, a tester or developer has to manually identify metamorphic relations using her knowledge of the program under test; this manual process can miss some important metamorphic relations that could reveal faults.

In this work we introduce a novel automated method for detecting metamorphic relations at the function level using a set of features extracted from the function itself. We model this problem as a machine learning classification problem. The automated method operates by extracting a set of features from a function's control flow graph and building a predictive model using machine learning techniques to classify whether a function exhibits a particular metamorphic relation or not. In addition, we show the effectiveness of these predicted metamorphic relations in detecting faults using mutation analysis.

Section II describes the metamorphic testing technique and the machine learning algorithms used in our work. Section III details our methodology. Section IV presents the evaluation of our method. Section V describes threats to validity. In Section VI we describe related work. We provide conclusions and future work in Section VII.

II. Background

A. Metamorphic Testing

Metamorphic testing supports the creation of follow-up test cases from existing test cases [4], [5] as follows:

```
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

1) Identify an appropriate set of metamorphic relations that the program under test should satisfy.
2) Create a set of *initial* test cases using techniques such as random testing, structural testing or fault based testing.
3) Create *follow-up* test cases by applying the input transformations required by the identified metamorphic relations in step 1 to each initial test case.
4) Execute the initial and follow-up test case pairs to check whether the output change complies with the change predicted by the metamorphic relation. A runtime violation of a metamorphic relation during testing indicates a fault or faults in the program under test.

Since metamorphic testing checks the relationship between inputs and outputs of multiple executions of the program under test, this method can be used when the correct result of individual executions are not known.

Consider the function in Figure 1 that calculates the sum of integers in an array *a*. Randomly permuting the order of the elements in *a* should not change the result. This is the permutative metamorphic relation in Table I. Further, adding a positive integer *k* to every element in *a* should increase the result by $k \times length(a)$. This is the additive metamorphic relation in Table I. Therefore, using these two relations, two follow-up test cases can be created for every initial test case and the outputs of the follow-up test cases can be predicted using the initial test case output.

Murphy et al. [9] identified the set of metamorphic relations in Table I that are common in mathematical functions. A function *f* is said to satisfy (or exhibit) a metamorphic relation *m* in Table I, if the change in the output after modifying the original input according to *m* can be predicted based on the original output. For example, if the input to *f* is modified by adding a positive constant, its output could either increase, decrease or remain without a change. A function *f* satisfies the additive metamorphic relation if we can predict the change in the output when the initial input is modified by adding a positive constant. In this work we apply machine learning techniques to automatically determine whether a function exhibits any of the metamorphic relations in Table I.

B. Machine Learning

Machine learning methods focus on providing the ability of computer programs to make better decisions based on experience [10]. Usually, the set of examples used by a machine learning algorithm are divided into two subsets: called a *training set* and a *test set*. The *training set* is used to create the predictive model, while the *test set* is used to evaluate the performance of the predictive model.

TABLE I
METAMORPHIC RELATIONS

Relation	Change made to the input
Additive	Add or subtract a constant
Multiplicative	Multiply by a constant
Permutative	Randomly permute the elements
Invertive	Take the inverse of each element
Inclusive	Add a new element
Exclusive	Remove an element
Compositional	Combining two or more inputs

Supervised learning is one machine learning method, where a set of labeled examples is used to learn a target function. The target function maps the input to a desired set of outputs (labels). Input to a supervised classification algorithm is a set of training data $S = \{s_1, s_2, ..., s_n\}$. Each vector $s_i = x_1, x_2, ..., x_m, c_i$ is called a *training instance*, where x_j is a *feature* and c_i is the class label of the training instance s_i. A *feature* is a measurable property of an instance.

In this work, we model metamorphic relation prediction as a supervised learning problem. For a given metamorphic relation we create a supervised classification model using features extracted from a set of functions already known to satisfy or not satisfy the considered metamorphic relation. Then the trained classification model is used to predict whether a previously unseen function will satisfy the considered metamorphic relation or not. Since we focus on predicting whether a function *f* exhibits a metamorphic relation *m* or not, this problem can be treated as a binary classification problem, in which the class label can take only one of two possible values ($+1/-1$). We used *Decision Trees* [11] and *Support Vector Machines (SVMs)* [12] as the classification algorithms.

1) Decision Trees (DT): In decision tree learning, the target function is a decision tree. In classification, a decision tree maps the input to a binary label. Internal nodes of a decision tree test a feature in the input and leaf nodes assign a label. We used the J48 Java implementation of the C4.5 [13] decision tree generation algorithm, from the WEKA [14] tool kit. When choosing a feature for an internal node, C4.5 chooses the feature with the highest information gain [15].

2) Support Vector Machines (SVMs): SVM [12] is another supervised learning algorithm that is used for classification. SVM creates a hyper-plane in a high dimensional space that can separate the instances in the training set according to their class labels. When a linear separation cannot be found in the original feature space, SVMs use *kernel functions* to map the training data into a higher dimensional feature space. Then the SVM creates a linear separator in this higher dimensional feature space, which can be used to classify unseen data instances. In this work we used the SVM implementation in the PyML Toolkit[1].

III. PROPOSED METHOD

Figure 2 shows an overview of our method. We start by creating the *control flow graph (CFG)* from a function's source

[1] http://pyml.sourceforge.net/

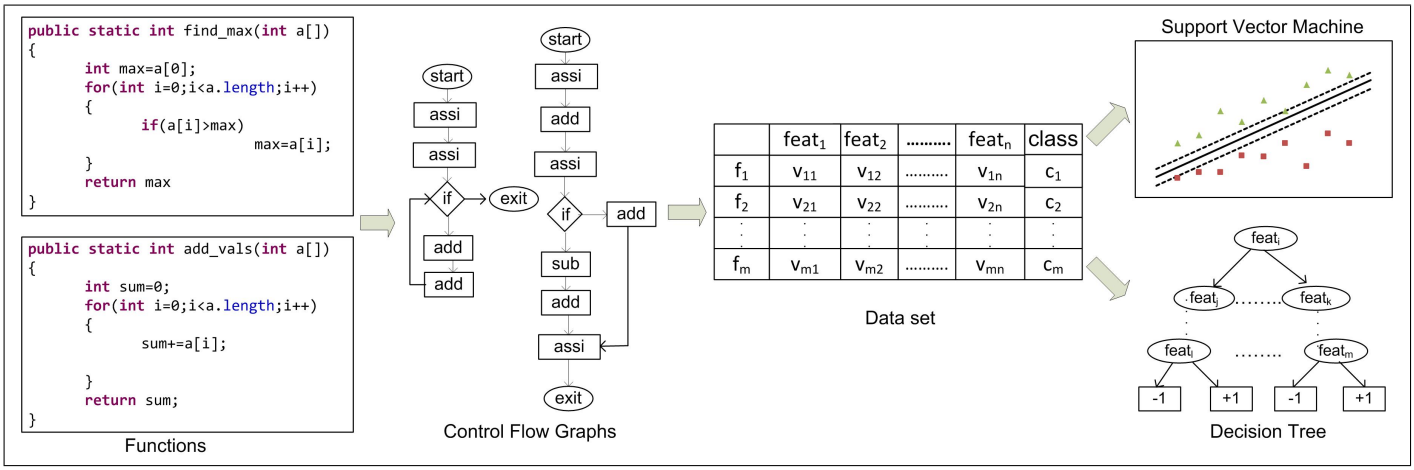


Fig. 2. Overview of the proposed method

code. Next, we extract a set of features from the CFGs, and a machine learning algorithm uses these features to create a predictive model. Finally, we use the developed predictive model to predict the metamorphic relations in previously unseen functions.

A. Function Representation

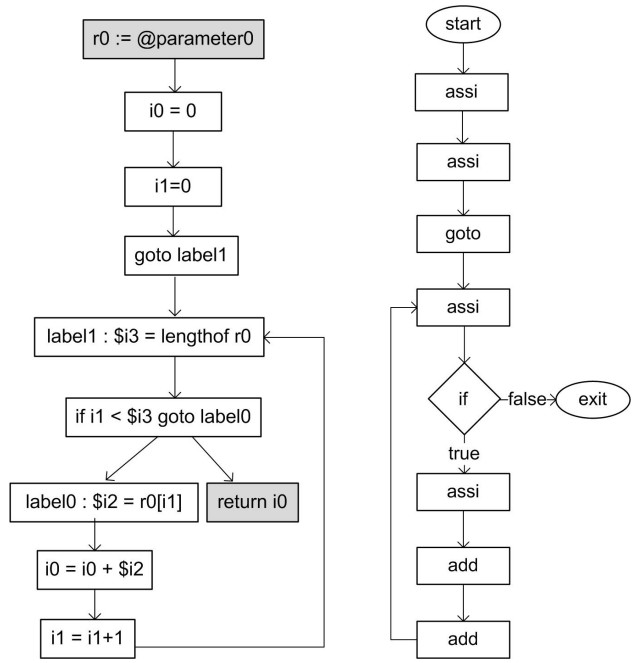
We hypothesize that the metamorphic relations in Table I are related to the sequence of operations performed by a function. Therefore, we represent a function using a statement level CFG, since it models the sequence of operations. The CFG $G = (N, E)$ of a function f is a directed graph, where each $n_x \in N$ represents a statement x in f and $E \subseteq N \times N$. An edge $e = (n_x, n_y) \in E$ if x, y are statements in f and y can be executed immediately after executing x. Nodes $n_{start} \in N$ and $n_{exit} \in N$ represents the starting and exiting points of f.

We used the *Soot*[2] framework to create CFGs. *Soot* generates control flow graphs in *Jimple* [16], a typed 3-address intermediate representation, where each CFG node represents an atomic operation. This representation should considerably reduce the effects of different programming styles and models the actual control flow structure of the function. Figure 3a is the *Jimple* statement level CFG generated using the *Soot* framework for the function in Figure 1. Converting Java code to the Jimple 3-address intermediate representation would add goto operations and labels to represent conditional jumps in the original Java code. Then a *labeled CFG* is created by giving a label to each node in the CFG in Figure 3a to indicate the operation performed in the node. Figure 3b is the labeled CFG created from the original CFG in Figure 3a.

B. Feature Extraction

We extracted two types of features based on the nodes and paths in the CFG.

[2]http://www.sable.mcgill.ca/soot/



(a) *Soot* CFG (b) Labeled CFG

Fig. 3. CFG generated by *Soot* with 3-address code and Labeled CFG for the program in Figure 1

1) Node Features: For a CFG, node features have the form $op - d_{in} - d_{out}$, where op is the operation performed in a node $n \in N$, d_{in} is the *in-degree* of n and d_{out} is the *out-degree* of n. The value for a given feature is the number of occurrences of nodes of type $op - d_{in} - d_{out}$ in the CFG. Table II shows the node features calculated for the labeled graph in Figure 3b.

2) Path Features: Features based on paths are created by taking the sequence of nodes in the shortest path from N_{start} to each node and the sequence of nodes in the shortest path from each node to N_{exit}. A path feature takes the form $op_1 - op_2 - ... - op_k$ where $op_i (1 \leq i \leq k)$ represents the operation performed in the CFG nodes in the considered path. The value

TABLE II
NODE FEATURES CALCULATED FROM THE LABELED GRAPH IN FIGURE 3B

Feature	Feature Value
start-0-1	1
if-1-2	1
add-1-1	2
assi-1-1	3
assi-2-1	1
goto-1-1	1
exit-1-0	1

of a path feature is the number of occurrences of that shortest path node sequence in the CFG. Table III shows the set of features extracted from the labeled graph in Figure 3b. For the example considered here, each node sequence occurs only once in the labeled graph in Figure 3b. Therefore each feature value for this example takes the value one.

TABLE III
PATH FEATURES CALCULATED FROM THE LABELED GRAPH IN FIGURE 3B

Feature	Feature Value
start-assi-assi-goto-assi-if-assi-add	1
start-assi-assi-goto-assi-if-assi	1
start-assi-assi	1
start	1
start-assi-assi-goto-assi-if	1
start-assi-assi-goto-assi-if-exit	1
start-assi-assi-goto-assi	1
start-assi	1
start-assi-assi-goto-assi-if-assi-add-add	1
start-assi-assi-goto	1
assi-goto-assi-if-exit	1
exit	1
goto-assi-if-exit	1
assi-if-exit	1
if-exit	1
assi-add-add-assi-if-exit	1
add-assi-if-exit	1
assi-assi-goto-assi-if-exit	1
add-add-assi-if-exit	1

C. Prediction

In this work we focus on predicting whether a given function f exhibits a metamorphic relation in Table I. We selected the *permutative*, *additive* and *inclusive* metamorphic relations for the initial experiment. These relations represent three different categories of input modifications: (1) changing the order of elements (permutative), (2) changing the element values (additive, multiplicative and invertive) and (3) adding/removing new element/s to/from the input (inclusive, exclusive and compositional). The three selected metamorphic relations represent a diverse initial set of relations for use in evaluating our method. Although there could be a variety of changes in the output when the input is modified using these relations, we have considered only the specific output changes given in Table IV for each metamorphic relation.

We modeled this problem as a machine learning classification problem, where each function f has a class label with the value 1 or -1 depending on whether f exhibits a specific metamorphic relation or not, respectively. Table V depicts an example data set used for learning the classifier, where f_i represents a function in the data set and $feat_j$ represents a node or path feature extracted from the labeled CFGs of functions. The feature value of $feat_j$ for the function f_i is represented by v_{ij} ; c_i represents the class label for the function f_i indicating whether f_i exhibits a specific metamorphic relation or not. Three data sets were created for each metamorphic relation in Table IV and they were used as input to the SVM and decision tree classification algorithms.

TABLE IV
METAMORPHIC RELATIONS AND ANTICIPATED CHANGES TO THE OUTPUT

Relation	Output change
Permutative	Remains constant
Additive	Remains constant or Increase
Inclusive	Remains constant or Increase

TABLE V
EXAMPLE DATA SET USED FOR PREDICTION

Function	$feat_1$	$feat_2$	...	$feat_n$	Class
f_1	v_{11}	v_{22}	...	v_{1n}	c_1
f_2	v_{11}	v_{22}	...	v_{2n}	c_2
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
f_m	v_{m1}	v_{m2}	...	v_{mn}	c_m

IV. EVALUATION

A. Data set

To measure the effectiveness of our proposed method, we built a code corpus containing 48 mathematical functions that take numerical inputs and produce numerical outputs. None of these functions have an oracle to check the correctness of the output for a randomly generated input. Table VI shows the details of the functions used in the experiment. These functions were implemented using the Java programming language.

B. Evaluation Procedure

Our evaluation procedure is two fold. We measured the effectiveness of (1) our predictive model and (2) the predicted metamorphic relations in detecting faults. The latter was conducted to validate the usefulness of the metamorphic relations predicted by our method.

1) Predictive Model Evaluation: We used *accuracy*, the *area under the receiver operating characteristic curve (AUC)* and *false positive rate (FPR)* as performance measures to evaluate the predictive models. *Accuracy* is the percentage of correct predictions made by the predictive model. *AUC* 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 [17]. So a higher AUC value indicates that the model has a higher predictive ability. Further, AUC is a more reasonable estimate than accuracy

TABLE VI
DETAILS OF THE FUNCTIONS USED IN THE EXPERIMENT (P: PERMUTATIVE, A: ADDITIVE, I: INCLUSIVE, ✓: POSITIVE EXAMPLE FOR THE RELATION, ×: NEGATIVE EXAMPLE FOR THE RELATION, -: NOT USED AS AN EXAMPLE FOR THE RELATION)

No.	Function Name	P	A	I
1.	*add_values* (Add elements in an array)	✓	✓	✓
2.	*add_two_array_values* (Adds elements at given index in 2 arrays)	×	✓	✓
3.	*bubble_sort* (Implements bubble sort)	✓	✓	×
4.	*shell_sort* (Implements of shell sort)	✓	✓	×
5.	*binary_search*	×	✓	✓
6.	*sequential_search*	×	✓	✓
7.	*selection_sort* (Implements selection sort)	✓	✓	×
8.	*dot_product*	×	✓	✓
9.	*array_div* (Divide array elements by *k*)	×	✓	-
10.	*set_min_val* (Set array elements less than *k* equal to *k*)	×	×	✓
11.	*find_min* (Find minimum value in an array)	✓	✓	✓
12.	*find_diff* (Element-wise difference in two arrays)	×	✓	-
13.	*array_copy* (Deep copy an array)	×	-	✓
14.	*copy_array_part*	×	-	✓
15.	*find_euc_dist* (Euclidean distance between two vectors)	×	✓	✓
16.	*find_magnitude* (magnitude of a vector)	✓	✓	✓
17.	*manhattan_dist* (Manhattan distance between two vectors)	×	×	✓
18.	*average*	✓	✓	✓
19.	*dec_array* (Decrement elements by *k*)	×	✓	-
20.	*find_max* (Find the maximum value)	✓	✓	✓
21.	*find_max2* (maximum value of addition of two consecutive elements in an array)	×	✓	×
22.	*quick_sort* (Implements quick sort)	✓	✓	×
23.	*variance*	✓	✓	×
24.	*insertion_sort* (Implements insertion sort)	✓	✓	×
25.	*heap_sort* (Implements heap sort)	✓	✓	×
26.	*merge_sort* (Implements of merge sort)	✓	✓	×
27.	*geometric_mean*	✓	✓	×
28.	*mean_absolute_error*	×	×	✓
29.	*select_k* (Find the k^{th} largest value from a set of numbers)	✓	✓	×
30.	*find_median*	✓	✓	×
31.	*cartesian_product* (Cartesian product between two sets)	✓	×	✓
32.	*reverse* (Reverse an array)	×	-	-
33.	*check_equal_tolerance* (Element-wise equality within a given tolerance in two sets of real numbers)	×	×	-
34.	*check_equal* (Element-wise equality between two sets of integers)	×	×	-
35.	*weighted_average*	×	✓	✓
36.	*count_k* (Occurrences of *k* in an array)	✓	×	✓
37.	*bitwise_and*	×	-	-
38.	*bitwise_or*	×	-	-
39.	*bitwise_xor*	×	-	-
40.	*bitwise_not*	×	-	-
41.	*clip* (Values outside a given interval clipped to the edges of the interval in an array)	×	×	-
42.	*elementwise_max* (Element-wise maximum)	×	×	-
43.	*elementwise_min* (Element-wise minimum)	×	×	-
44.	*cnt_nzeroes* (Number of non-zero elements in an array)	✓	×	✓
45.	*cnt_zeros* (Number of zero elements in a given array	✓	×	✓
46.	*elementwise_equal* (Check for element-wise equality and returns a boolean array)	×	×	-
47.	*elementwise_not_equal* (Check two for element-wise for nonequality and returns a boolean array)	×	×	-
48.	*hamming_dist*	-	×	-

when comparing the performance of predictive models since AUC is statistically consistent and more discriminating than accuracy [18], [19]. 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. A classifier with $AUC > 0.9$ is considered to be a highly effective classifier in the machine learning community.

FPR is the percentage of negative examples that were classified as positive.

$$FPR = \frac{false\ positives}{false\ positives + true\ negatives}$$

A classifier used for predicting metamorphic relations should have a low FPR, since incorrectly classifying a program to satisfy a specific metamorphic relation would ultimately result in investing resources to find a fault that is not actually present in the program.

We used *stratified k-fold cross-validation* to evaluate the performance of classification. 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 to build the predictive model (training) and the remaining subset is used to evaluate the performance of the predictive

TABLE VII
SUMMARY OF THE DATA SETS USED FOR PREDICTION

Metamorphic Relation	#Positive	#Negative	Total
Permutative	20	21	41
Additive	21	15	36
Inclusive	15	12	27

model (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 away that the folds contain approximately the same proportion of positive (functions that exhibit a specific metamorphic relation) and negative (functions that do not exhibit a specific metamorphic relation) examples as in the original data set.

2) Fault Detection Effectiveness: We used mutation analysis [20] to measure the effectiveness of the predicted metamorphic relations from our method in detecting faults. Mutation analysis operates by inserting faults into the program under test such that the created faulty version is very similar to the original version of the program [21]. A faulty version of the program under test is called a *mutant*. If a test identifies a mutant as faulty that mutant is said to be *killed*.

Mutation analysis was conducted on 35 functions from Table VI, which exhibits all or several of the three relations in Table IV. We used the μJava[3] mutation engine to create the mutants for the functions in our code corpus. We used only method level mutation operators [22] to create mutants since we are only interested in the faults at the function level. Each mutated version of a function was created by inserting only a single mutation. Mutants that resulted in compilation errors, run-time exceptions or infinite loops were removed before conducting the experiment.

For each function f we randomly generated 10 initial test cases. We then created follow-up test cases using the metamorphic relations predicted for f, for each of the initial test cases. Finally we checked whether the corresponding metamorphic relations were satisfied by the initial and follow-up test case pairs. A mutant of f is killed, if at least one pair of test cases fail to satisfy the corresponding metamorphic relation m.

C. Predictive Model Evaluation Results

We conducted the evaluation of the prediction of the three metamorphic relations in Table IV. Table VII gives the number of positive and negative examples contained in each of the data sets used for the evaluation. Using imbalanced proportions of data to train a classifier may result in biases in the classification algorithm [23]. Therefore, as reported in Table VII, each data set was created using approximately 50% positive and negative examples taken from the functions in Table VI.

Table VIII reports the *Accuracy* and *AUC* achieved by the SVM and decision tree models respectively. Results in Table VIII were obtained by applying *stratified 6-fold cross validation* 10 times using a different seed each time. Consequently, data is partitioned differently into the six folds for the 10 runs. This gives better performance measures than conducting the cross validation procedure one time. The p-values are calculated using the paired t-test. Reported values in the Table VIII are the average accuracies and AUC values for the 10 cross validation runs with the standard deviation reported inside parenthesis. Results for the SVM were obtained using the linear kernel and the default parameters (C=10) provided by PyML. As shown in Figure 4 and 5, when compared with the decision tree model, SVM gives an overall better performance with respect to accuracy and AUC measures for all three metamorphic relations. As shown by the p-values reported in Table VIII the difference in performance between SVMs and decision trees are statistically significant at the 0.05 significance level. This is expected because SVMs tend to perform better than other machine learning methods since they are less prone to overfitting [24]. For all the three relations used for prediction, SVM achieves an accuracy higher than 0.8 and AUC higher than 0.9. The high accuracy and AUC achieved by the SVM model for the three metamorphic relations shows that the feature set developed using the CFGs can effectively predict metamorphic relations.

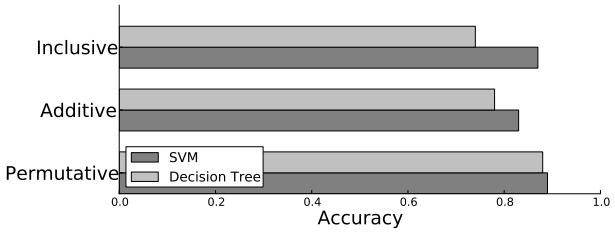


Fig. 4. Accuracy for SVM and Decision Tree models for predicting Permutative, Additive and Inclusive Metamorphic Relations

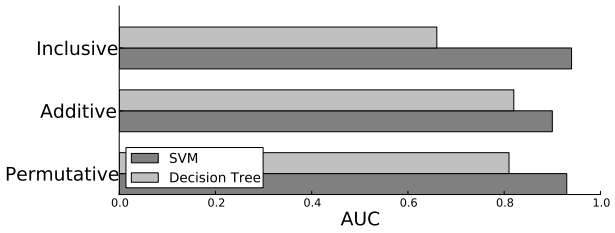


Fig. 5. AUC for SVM and Decision Tree models for predicting Permutative, Additive and Inclusive Metamorphic Relations

Table IX shows the FPR for SVMs and decision trees when predicting the three metamorphic relations. When predicting permutative and additive metamorphic relations SVMs and decision trees does not show a significant difference of FPR at the 0.05 significance level. But when predicting the inclusive metamorphic relation, SVMs achieve a significantly low FPR than decision trees.

Secondly, we evaluated how the performance of a classifier varies with the training set size when predicting metamorphic

[3]http://cs.gmu.edu/∼offutt/mujava/

TABLE VIII
PREDICTION RESULTS FROM SVMS AND DECISION TREES

Metamorphic Relation	Accuracy			AUC		
	SVM	DT	p-value	SVM	DT	p-value
Permutative	0.89 (0.04)	0.87 (0.03)	8.30E-14	0.93 (0.03)	0.81 (0.03)	6.37E-08
Additive	0.83 (0.04)	0.78 (0.05)	1.27E-11	0.90 (0.04)	0.82 (0.04)	1.35E-03
Inclusive	0.87 (0.02)	0.73 (0.02)	1.19E-14	0.94 (0.03)	0.66 (0.03)	1.51E-09

TABLE IX
FPR FOR SVMS AND DECISION TREES

Metamorphic Relation	SVM	DT	p-value
Permutative	0.10	0.10	1.00
Additive	0.22	0.26	0.46
Inclusive	0.22	0.35	9.00E-5

relations. Using a large number of examples to train a classifier will reduce the chance of the classifier being biased [25]. But using a large training set will increase the cost of this approach since it requires identifying metamorphic relations for the programs in the training set manually. Therefore we plan to investigate whether effective classifiers can be learned using a small set of programs.

We used SVMs in this evaluation since they performed significantly better than the decision trees. We first randomly partitioned each dataset in Table VII into two sets so that each half would contain approximately the same proportions of positive and negative examples as the original dataset. One partition was used as the test set. From the other partition we randomly selected training examples to train the classifier. For each training set size we tested the trained classifier using all instances in the test set.

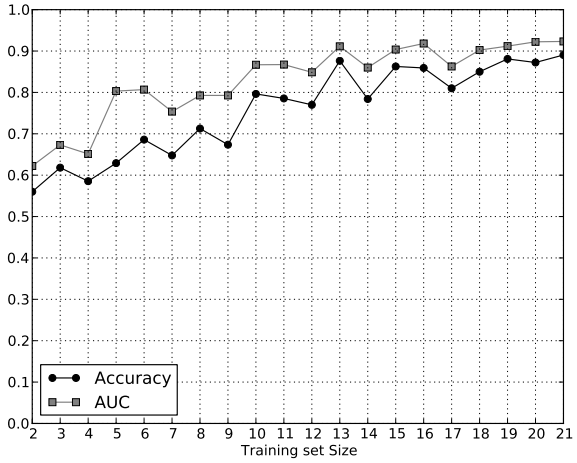


Fig. 6. Variation of accuracy and AUC with training set size for predicting permutative MR

Figures 6, 7 and 8 show the variation of the accuracy and AUC of the classifiers for different training set sizes in predicting permutative, additive and inclusive metamorphic relations respectively. As expected, the accuracy and AUC of

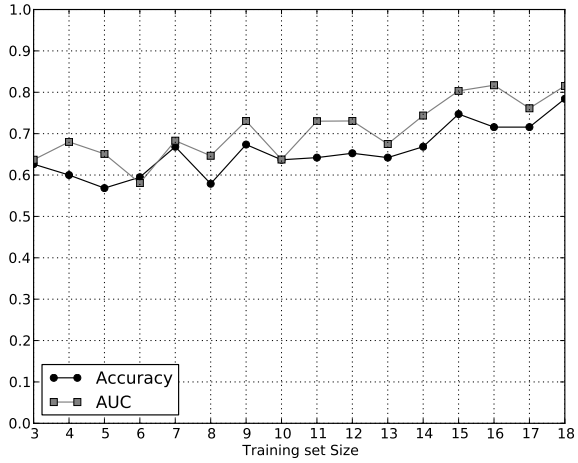


Fig. 7. Variation of accuracy and AUC with training set size for predicting additive MR

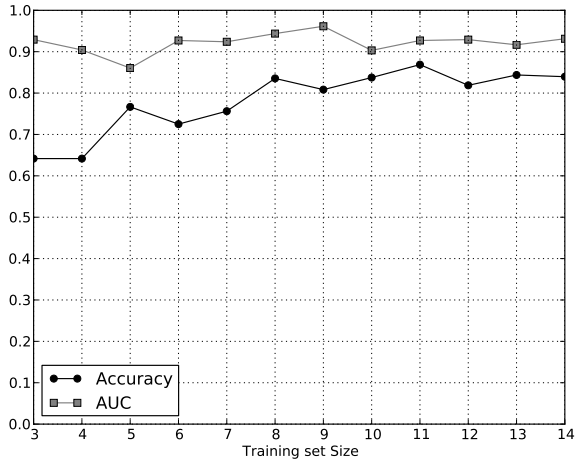


Fig. 8. Variation of accuracy and AUC with training set size for predicting inclusive MR

the classifiers increase with the training set size. For the three metamorphic relations, even classifiers built using the smallest possible training set perform better than a random classifier. For the permutative relation, the classifier achieves a 0.8 AUC when trained with a set of five programs. For the inclusive relation, even the smallest training set containing only three programs achieved an AUC of 0.8. For the additive relation,

a 0.8 AUC was achieved with a training set of 15 programs. These result show that our method works effectively with a small number of training examples.

Finally we evaluated the performance of the classifiers when the prediction is done on programs that contain at least one fault. In practice, a test engineer may have a set of programs that satisfies known metamorphic relations. The engineer can apply our prediction method to determine which of these metamorphic relations should be satisfied by a new, possibly faulty program. To evaluate how our method works when applied to such a scenario, we did the following for each function f in the data set:

1) We removed f from the data set and trained the SVM classifier using the remaining functions.
2) We generated predictions for the original function f and for randomly selected mutants of function f using the classifier produced in Step 1.
3) We compared the classification for each mutant to the classification for the original function.

Table X shows the results of this evaluation. The third column shows the number of mutants that were classified differently than the original function. The classification of the mutants did not match that of the original function for only 1%-5% of the mutants, depending on the metamorphic relation. These results indicate that the prediction model can provide a reasonably accurate classification for functions that contain a fault.

TABLE X
RESULTS OF PREDICTING METAMORPHIC RELATIONS FOR THE FAULTY VERSIONS OF THE FUNCTIONS

Metamorphic Relation	# Mutants used	# Classified differently
Permutative	171	3
Additive	146	7
Inclusive	131	2

D. Fault Detection Effectiveness Results

Table XI gives a summary of the mutants generated for the mutation analysis performed to evaluate the effectiveness of our approach in detecting faults. After removing mutants that are obviously incorrect (mutants that gave exceptions and infinite loops), 988 mutants in total were used for the experiment.

Out of 988 mutants, 655 (66%) were killed using the predicted metamorphic relations. More than 50% of the mutants were killed in 29 functions. This shows that we can apply these predicted relations successfully to detect faults. Table XII shows the percentages of mutants that were killed using each metamorphic relation alone. The permutative relation has the highest percentage of killed mutants. This result was expected since, for the functions studied in this experiment, the permutative metamorphic relation requires the outputs of the initial and the follow-up test cases to be equal (see Table IV). This equality relation is more restrictive and thus can be more easily violated than the inequality relation of the additive and inclusive metamorphic relations.

TABLE XI
SUMMARY OF MUTANTS USED IN THE MUTATION ANALYSIS

# Mutants generated by muJava	1717
# Mutants resulted in Exceptions	591
# Mutants resulted in Infinite loops	138
# Mutants used in the experiment	988

TABLE XII
PERCENTAGE OF MUTANTS KILLED BY EACH METAMORPHIC RELATION

Metamorphic Relation	# Mutants possible to kill	# Mutants killed (%)
Permutative	566	374 (66%)
Additive	869	196 (23%)
Inclusive	400	150 (38%)

Several mutants could not be killed using any of the predicted metamorphic relations. Some of these mutants are making changes that could not be captured by the metamorphic relations that we used in this study. Additional metamorphic relations might be needed to kill them. Some of the survivors are equivalent mutants that cannot be killed since they produce the same output as the original program [21].

V. THREATS TO VALIDITY

The main threat to external validity is the limited set of programs studied in this experiment. We used 48 mathematical programs that implement functions commonly used in scientific programs. While these programs do not necessarily represent all the programs without oracles, they perform a variety of functionalities such as sorting, searching, calculating common statistics, etc. Further, we focus on the problem of predicting the likely metamorphic relations at the unit level. Therefore, we used a set of programs that implements a single functionality. Even though we only used a set of small mathematical functions, the results demonstrate the potential effectiveness of this novel approach for detecting likely metamorphic relations. The path features contain the shortest path from one node to every other node in the graph. So, the features include information about sequences of nodes that are fragments of paths. Therefore even if the training set does not contain information regarding the entire path of a larger function, the classifier should produce the correct prediction using the information about the path fragments. Therefore our approach should generalize to large functions even if the model is not trained with functions of the same size. In fact, functions used in the experiment vary in size between 7 to 45 LOC.

The main threat to internal validity concerns the faults in the implementations of these functions. We use an abstract representation of the functions to teach the classifier. Based on the competent programmer hypothesis [26], even though the programmer might make some mistakes in the implementation, the general structure of the program should be similar to the fault-free version of the program. In addition, there may be more relevant metamorphic relations and/or program features for the programs studied.

The main threat to conclusion validity is the sample size used in the validation. We used 48 programs that take numerical inputs and produce numerical outputs in this study. We limited the set of programs to 48 since we believe it is not cost effective for the classifier to learn from a large set of programs.

We used the *Soot* framework to generate CFGs of the functions used in this experiment. Further we used the *NetworkX*[4] package for graph manipulation. Usage of these third party tools represents potential threats to construct validity. We verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool. Further, we used mutation analysis to measure the fault detection effectiveness of predicted metamorphic relations. Mutation analysis represents a threat to construct validity because mutations are synthetic faults. However, previous studies have shown that mutations represent faults made by a real human programmer [20].

VI. Related Work

To our knowledge, this is the first time that machine learning techniques have been used to predict metamorphic relations. Murphy suggested using machine learning to detect likely metamorphic relations [27]. Hasan [28] manually identified several code patterns that exhibit some of the metamorphic relations in Table I and proposed a method based on data flow analysis to identify those patterns within code, but did not evaluate the proposed method.

Metamorphic testing has been used to test applications without oracles in different areas. Xie et al. used metamorphic testing to test machine learning applications [7]. Metamorphic testing was used to test simulation software such as health care simulations [29] and Monte Carlo modeling [30]. Metamorphic testing has been used effectively in bioinformatics [31], computer graphics [32] and for testing programs with partial differential equations [33]. Murphy et al. [34] show how to automatically convert a set of metamorphic relations for a function into appropriate test cases and check whether the metamorphic relations hold when the program is executed. Murphy et al. specify the metamorphic relations manually.

Metamorphic testing has also been used to test programs at the system level. Muphy et al. developed a method for automating system level metamorphic testing [35]. In this work, they also describe a method called *heuristic metamorphic testing* for testing applications with non-determinism. All of these approaches can benefit from our method for automatically finding likely metamorphic relations.

Machine learning, specifically decision trees and SVMs, have been used in different areas of software testing. For example, data collected during software testing involving test case coverage and execution traces can potentially be used in fault localization, test oracle automation, etc. [36]. Bowring et al. used program execution data to classify program behavior [37]. Briand et al. [38] used the C4.5 machine learning algorithm for test suite refinement. Briand et al. [39] used machine learning for fault localization to reduce the problems faced by other fault localization methods when several faults are present in the code. They used the C4.5 machine learning algorithm to identify failure conditions, and determined if the failure occurred due to the same fault(s). Frounchi et al. [40] used machine learning to develop a test oracle for testing an implementation of an image segmentation algorithm. They used the C4.5 algorithm to build a classifier to determine whether a given pair of image segmentations are consistent or not. Once the classification accuracy is satisfactory, the classifier can check the correctness of the image segmentation program. Lo used a SVM model for predicting software reliability [41]. Wang et al. [42] used SVMs for creating test oracles for reactive systems.

[4]http://networkx.lanl.gov/

VII. Conclusions and Future Work

We have presented a novel machine learning approach to automatically detect likely metamorphic relations of program functions using features extracted from a function's control flow graph. We have evaluated the performance of our predictive model using a set of real world functions that do not have oracles. Overall, SVMs performed significantly better than decision trees. High accuracy and AUC achieved by the SVM predictive model show that features developed using the CFGs are highly effective in predicting metamorphic relations. In addition, our method reports a low FPR making it suitable for predicting likely metamorphic relations. We also showed that our method can create effective classifiers using reasonably small training sets making this approach cost effective for use in practice. Further, we show that, when applied to programs with an injected fault, our method produces the same predictions as those produced for the original program in at least 95% of the cases. Thus, the identified metamorphic relations should be accurate even for faulty programs. Finally, using mutation analysis we showed that the predicted relations can effectively detect faults.

By automatically detecting metamorphic relations, our approach is a contribution towards reaching the goal of making non-testable software testable [1]. Our approach will provide the ability to fully automate the metamorphic testing process, which will help to reduce the testing cost. This approach is applicable to many scientific applications as well as other programs without test oracles.

In the future, we plan to conduct experiments to evaluate the effectiveness of our method in predicting the remaining four metamorphic relations in Table I. In addition, we plan to evaluate the effectiveness of features based on other aspects of the program such as the data flow and predicates for predicting metamorphic relations. Further, we plan to investigate the possibility of using a multi-label classifier instead of creating binary classifiers for predicting individual metamorphic relations. Since the best results were achieved by the SVMs, we plan to investigate the application of SVMs that use graph kernels that can directly measure the similarity between graphs in different aspects. We have started applying the random walk

graph kernel [43] in the context of predicting metamorphic relations.

REFERENCES

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

[2] R. Sanders and D. Kelly, "The challenge of testing scientific software," in *Proc. Conf. for the Association for Software Testing (CAST)*, Toronto, July 2008, pp. 30–36.

[3] L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'Hara, D. Gavaghan, and S. Emmott, "Troubling trends in scientific software use," *Science*, vol. 340, no. 6134, pp. 814–815, 2013.

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

[5] 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.

[6] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," in *Proc. of the 8th Int. Symp. on Future Software Technology (ISFST 2004)*. Software Engineers Association, 2004.

[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] 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 *Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, 2004, pp. 569–583.

[9] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *Proc. of the 20th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA*, 2008, pp. 867–872.

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

[11] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[12] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

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

[14] 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.

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

[16] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," 1998.

[17] 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.

[18] F. Provost, T. Fawcett, and R. Kohavi, "The case against accuracy estimation for comparing induction algorithms," in *Proc. of the 15th Int. Conf. on Machine Learning*, 1997, pp. 445–453.

[19] C. X. Ling, J. Huang, and H. Zhang, "AUC: a better measure than accuracy in comparing learning algorithms," in *Proc. of IJCAI03*, 2003, pp. 329–341.

[20] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of the 27th Int. Conf. on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 402–411.

[21] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, 2011.

[22] Y.-S. Ma and J. Offutt, "Description of method-level mutation operators for java," November 2005.

[23] G. M. Weiss, "Mining with rarity: a unifying framework," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 7–19, Jun. 2004.

[24] R. Burbidge and B. Buxton, "An introduction to support vector machines for data mining," in *Keynote Papers, Young OR12, University of Nottingham, Operational Research Society*, 2001, pp. 3–15.

[25] S. Beiden, M. Maloof, and R. Wagner, "A general model for finite-sample effects in training and testing of competing classifiers," *Pattern Analysis and Machine Intelligence, IEEE Trans on*, vol. 25, no. 12, pp. 1561 – 1569, Dec 2003.

[26] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34 –41, april 1978.

[27] C. Murphy, "Metamorphic testing techniques to detect defects in applications without test oracles," PhD dissertation, Columbia University, 2010.

[28] S. Hasan, "Automatic detection of metamorphic properties of software," Columbia University, Tech. Rep. CUCS-003-12, 2009.

[29] 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 *Proc. of the 3rd Workshop on Software Engineering in Health Care*, ser. SEHC '11. New York, NY, USA: ACM, 2011, pp. 40–47.

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

[31] 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.

[32] R. Guderlei and J. Mayer, "Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *Proc. of the 7th Int. Conf. on Quality Software*, ser. QSIC '07, 2007, pp. 404 –409.

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

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

[35] ——, "Automatic system testing of programs without test oracles," in *Proc. of the 18th Int. symp. on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 189–200.

[36] L. C. Briand, "Novel applications of machine learning in software testing," in *Proc. of the 2008 The 8th Int. Conf. on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–10.

[37] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Proc. of the 2004 ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 195–205.

[38] L. C. Briand, Y. Labiche, and Z. Bawar, "Using machine learning to refine black-box test specifications and test suites," in *Proc. of the 8th Int. Conf. on Quality Software*, ser. QSIC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 135–144.

[39] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Software Reliability, 2007. ISSRE '07. The 18th IEEE Int. Symposium on*, 2007, pp. 137 –146.

[40] 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.

[41] J.-H. Lo, "Predicting software reliability with support vector machines," in *Computer Research and Development, 2010 2nd Int. Conf. on*, may 2010, pp. 765 –769.

[42] F. Wang, L.-W. Yao, and J.-H. Wu, "Intelligent test oracle construction for reactive systems without explicit specifications," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE 9th Int. Conf. on*, dec. 2011, pp. 89 –96.

[43] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Learning Theory and Kernel Machines*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2777, pp. 129–143.