SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2014; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach Using Graph Kernels

Upulee Kanewala*, James M. Bieman, and Asa Ben-Hur

Computer Science Department, Colorado State University, USA

SUMMARY

Comprehensive, automated software testing requires an oracle to check whether the output produced by a test case matches the expected behavior of the program. But the challenges in creating suitable oracles limit the ability to perform automated testing in some programs including scientific software.
Metamorphic testing is a method for automating the testing process for programs without test oracles. This technique operates by checking whether the program behaves according to a certain set of properties called *metamorphic relations*. A metamorphic relation is a relationship between multiple input and output pairs of the program. 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 conduct a series of empirical studies to evaluate the effectiveness of graph kernels, a mechanism to compute a similarity score between pair graphs. These graph kernels use a variety of substructures in graph based program representations for predicting metamorphic relations. In addition, we compare the effectiveness of program control flow and data dependency information for predicting metamorphic relations.
Results of our empirical studies show that graph kernels can improve the prediction effectiveness. In addition we show that incorporating domain information such as properties of mathematical operations improves prediction effectiveness of the graph kernels. Further, we show that program control flow information is more effective in predicting metamorphic relations than data dependency information. Finally, we show that combining control flow and data dependency information might be useful when predicting some metamorphic relations.

Received . . .

KEY WORDS: Metamorphic testing, Metamorphic relations, Graph kernels, Support vector machines

1. INTRODUCTION

Scientific software is widely used in science and engineering. Such software plays an important role in critical decision making in fields such as the nuclear industry, medicine and the military [1, 2]. In addition, results obtained from such software are used as evidence for research publications [2]. But

*Correspondence to: upuleegk@cs.colostate.edu due to the lack of systematic testing in scientific software, subtle faults can remain undetected. These subtle faults can cause incorrect program outputs without causing a program to crash. Software faults such as one-off errors caused loss of precision in seismic data processing programs [3]. Software faults compromised coordinate measuring machine (CMM) performance [4]. In addition, there have been several retractions of published work due to software faults [5]. Hatton et al. [6] found that several software systems written for geoscientists produced reasonable yet essentially different results. There were situations where scientists believed that they needed to modify the physics model or develop new algorithms but later discovered that the real problem was small faults in the code [7]. Therefore it is important to conduct comprehensive automated testing on scientific software to ensure that they are implemented correctly.

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" [8] where an oracle is unavailable or too difficult to implement. In such situations, a domain expert must manually check that the output produced from the application is correct for a selected set of inputs. Further, Sanders et al. [1] 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 faults such as one-off errors, and hinders the automation of the testing process. A recent survey conducted by Joppa et al. showed that when adopting scientific software, only 8% of the scientists independently validate the software and the others choose to use the software simply because it was published in a peer-reviewed journal or based on personal opinions and recommendations [9]. Therefore undetected subtle faults 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 [10].

Metamorphic testing is a technique, introduced by Chen et al. [11], 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 (MR)* specifies how a particular change to the input of the program should change the output [12]. For example, in a program that calculates the average of an integer array, randomly permuting the order of the elements in the input array should not change the result. This property can be used as a MR for testing this program.

Violation of a MR occurs when the change in the output differs from what is specified by the considered MR. Satisfying a particular MR does not guarantee that the program is implemented correctly. However, a violation of a MR indicates that the program contains faults. Previous studies show that metamorphic testing can be an effective way to test programs without oracles [12, 13]. Enumerating a set of MRs that should be satisfied by a program is a critical initial task in applying metamorphic testing [14, 15]. Currently, a tester or a developer has to manually identify MRs using her knowledge of the program under test; this manual process can easily miss some of the important MRs that could reveal faults.

In our previous work we showed that we can create machine learning prediction models that can automatically predict MRs of previously unseen functions using a set of features extracted from the control flow graph of the function [16]. We showed that these prediction models make consistent predictions even when the functions contain faults. In this work we extend our previous work by investigating:

1. The effectiveness of different substructures in graph based function representations for predicting MRs.
2. The effectiveness of features that would represent the control flow and data dependency information of a function for predicting MRs.

In addition we extend our empirical studies by adding new functions obtained from open source code libraries. We also use several new MRs that were not used in our previous study.

The remainder of this paper is organized as follows: Section 2 describes details about metamorphic testing and kernel methods. Our approach including the detailed information about how we apply graph kernels are described in Section 3. Section 4 and Section 5 describes our experimental setup and the results of our empirical studies, respectively. We discuss threats to validity in Section 6. Section 7 presents the related work. Section 8 provides our conclusions and future work.

2. BACKGROUND

In this section we present some background information about metamorphic testing and kernel methods we use in this work.

2.1. Metamorphic testing

Metamorphic testing was introduced by Chen et al. [11] to alleviate the oracle problem. It supports the creation of follow-up test cases from existing test cases [11, 12] through the following process:

1. Identify an appropriate set of MRs 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 MRs in Step 1 to each initial test case.
4. Execute the corresponding initial and follow-up test case pairs to check whether the output change complies with the change predicted by the MR. A run-time violation of a MR 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* MR 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* MR 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.

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

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

2.2. *Kernel methods*

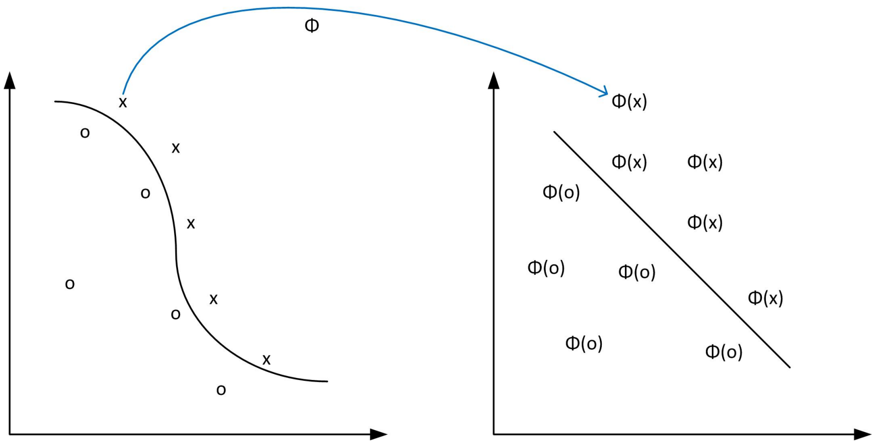


Figure 2. Function ϕ maps data into a new feature space such that a linear separation can be found.

Kernel methods perform pattern analysis by following two main steps: (1) Embed the data in an appropriate feature space. (2) Use machine learning algorithms to discover linear relations in the embedded data [17]. Figure 2 depicts how the data is mapped to a feature space such that a linear separation of the data can be obtained. There are two main advantages of using kernel methods. First, machine learning algorithms for discovering linear relations are efficient and are well understood. Second, a *kernel function* can be used to compute the inner product of data in the new feature space without explicitly mapping the data into that space. A *kernel function* computes the inner products in the new feature space directly from the original data [17]. Let k be the kernel function and $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m$ be training data. Then the kernel function calculates the inner product in the new feature space using only the original data without having to compute the mapping ϕ explicitly as follows: $k(\mathbf{x}_i, \mathbf{x}_j) = \langle\phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle$. Machine learning algorithms such as support vector machines (SVMs) can use the kernel function instead of calculating the actual coordinates in the new feature space.

In this work we represent the program functions using a graph that captures both control flow and data dependency information. Therefore we provide some background on methods used for comparing graphs. Graph comparison/classification has been previously applied in areas such as bioinformatics, chemistry, sociology and telecommunication. Graph comparison algorithms can be divided into three groups [18]: (1) *set based*, (2) *frequent subgraph based* and (3) kernel based. Set based methods compare the similarity between the set of nodes and the set of edges in two graphs. These methods do not consider the topology of the graph. Frequent subgraph based methods first develop a set of subgraphs that are frequently present in a graph dataset of interest. Then these selected subgraphs are used for graph classification. Frequent subgraph based methods are computationally expensive and the complexity increases exponentially with the graph size. In this paper we use the kernel based approach, specifically graph kernels and compare their performance to the set of features we used in our previous work [16].

Graph kernels are a set of kernel functions used for graph data. Graph kernels compute a similarity score for a pair of graphs by comparing their substructures, such as shortest paths [19], random walks [20], and subtrees [21]. Graph kernels provide a method to explore the graph topology by comparing graph substructures in polynomial time. Therefore graph kernels can be used efficiently to compare similarities between programs which are represented as graphs (e.g. control flow graphs and program dependency graphs).

3. APPROACH

Figure 3 shows an overview of our approach. During the *training phase*, we start by creating a graph based representation that shows both the control flow and data dependency information of the functions in the *training set*, which is a set of functions associated with a label that indicates if a function satisfies a given metamorphic relation (positive example) or not (negative example). Then we compute the graph kernel values that give a similarity score for each pair of functions in the training set. Then the computed graph kernel is used by an SVM to create a predictive model. During the *testing phase*, we use the developed model to predict whether a previously unseen function satisfies the considered metamorphic relation.

3.1. Function Representation

We used the following graph based representation of a function that contains both control flow information and data dependency information: $G_f = (V, E)$ of a function f is a directed graph, where each $v_x \in V$ represents a statement x in f. Each node is labeled with the operation performed in x, denoted by $label(v_x)$. An edge $e = (v_x, v_y) \in E$ if x, y are statements in f and y can be executed immediately after executing x. These edges represent the control flow of the function. An edge $e = (v_x, v_y) \in E$ if x, y are statements in f and y uses a value produced by x. These edges represent the data dependencies in the function. The label of an edge (v_x, v_y) is denoted by $label(v_x, v_y)$ and it can take two values: “cfg” or “dd” depending on whether it represents a control flow edge or a data dependency edge, respectively. Nodes $v_{start} \in V$ and $v_{exit} \in V$ represent the starting and exiting points of f [22].

We used the *Soot*[†] framework to create this graph based representation. *Soot* generates control flow graphs (CFG) in *Jimple* [23], a typed 3-address intermediate representation, where each CFG node represents an atomic operation. Consequently, each node in the graph is labeled with the atomic

[†]http://www.sable.mcgill.ca/soot/

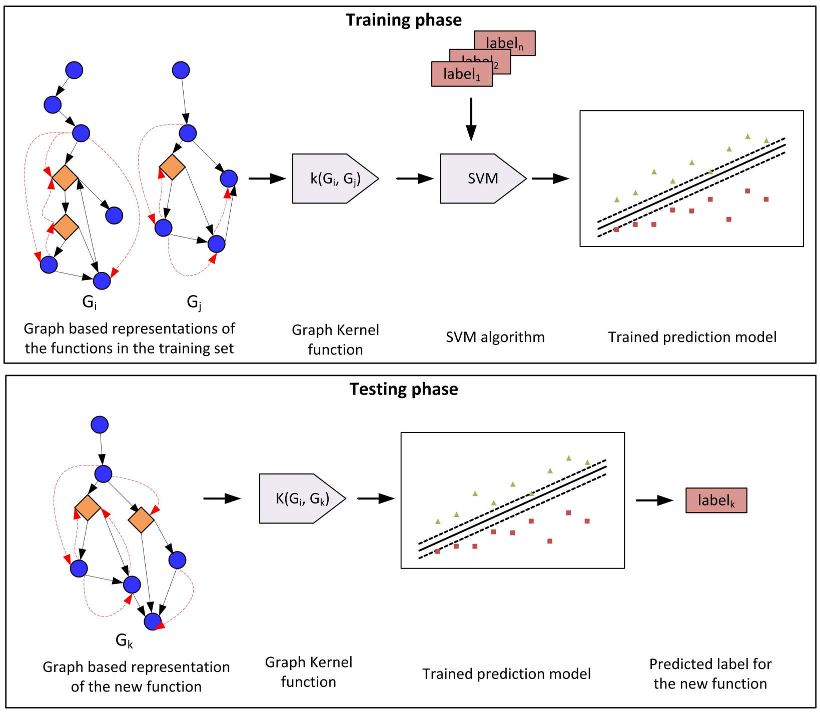


Figure 3. Overview of the approach. During the training phase a set of functions associated with a label representing the satisfiability of an MR is used for training an SVM classifier that use graph kernel values computed using the graph represenions of these functions. During the testing phase the trained model is used to predict whether a previously unseen function satisfies the MR.

operation performed. Then we compute the definitions and the uses of the variables in the function and use that information to augment the CFG with edges representing data dependencies in the function. Figure 4 displays the graph based representation created for the function in Figure 1.

3.2. Graph Kernels

We define two graph kernels for the graph representations of the functions presented in Section 3.1: the *random walk kernel* and the *graphlet kernel*. Each kernel captures different graph substructures as described next.

3.2.1. Random walk kernel Random walk graph kernels [24, 20] count the number of matching walks in two labeled graphs. In what follows we explain the idea, while the details are provided in Appendix A. The value of the random walk kernel between two graphs is computed by summing up the contributions of all walks in the two graphs. Each pair of walks is compared using a kernel that computes the similarity of each step in the two walks, where the similarity of each step is a product of the similarity of its nodes and edges. This concept is illustrated in Figure 5. Computing this kernel requires specifying an edge kernel and a node kernel. We used two approaches for determining the kernel value between a pair of nodes. In the first approach, we assign a value of one to the node kernel value if the two node labels are identical, and zero otherwise. In the second approach, we assign a value of 0.5, if the node labels represent two operations with similar properties, even if they

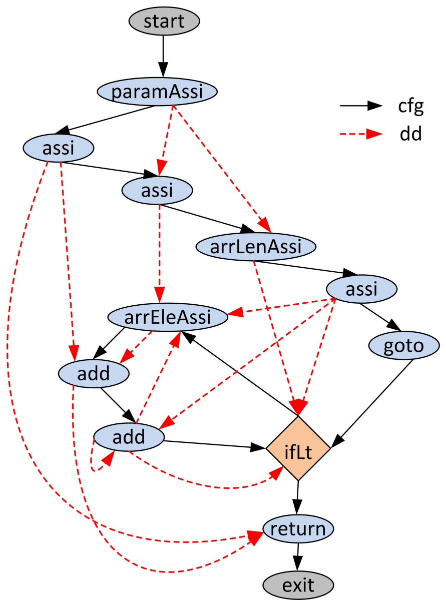


Figure 4. Graph representation of the function in Figure 1. cfg: control flow edges. dd: data dependency edges.

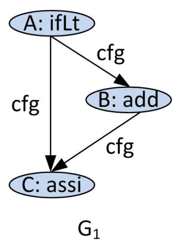
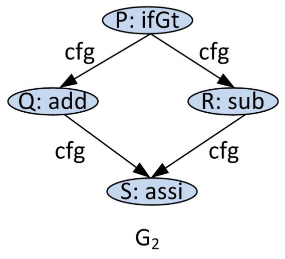
are not identifcal (Section A equation (5)). The kernel value between pair of edges is determined using their edge labels, where we assign a value of one if the edge labels are identical zero otherwise.

3.2.2. Graphlet kernel Random walks represent sequences of nodes of varying length and do not capture subgraphs in a graph. Subgraphs can directly capture the structure of *if conditions* in a function that represent important semantic information about the function. Therefore we apply a kernel based on subgraphs.

The graphlet kernel computes a similarity score of a pair of graphs by comparing all subgraphs of limited size in the two graphs [18]. In this work we use connected subgraphs with size $k \in \{3, 4, 5\}$ nodes. These subgraphs are called *graphlets*. Consider the pair of graphs G_3 and G_4 in Figure 6. The graphlet kernel value of a pair of graphs is calculated by summing up the kernel values of all the graphlet pairs in the two graphs. The kernel value of a pair graphlets is calculated as follows: for each pair of graphlets that are isomorphic, we compute the kernel value by multiplying the kernel values between the node and edge pairs that are mapped by the isomorphism function. If a pair of graphlets are not isomorphic, we assign a value of zero to the kernel value of those two graphlets. The kernel value between pairs of nodes and pairs of edges are determined as explained in Section 3.2.1. In Figure 6 we illustrate the computation of the kernel and the complete definition of the graphlet kernel is presented in Appendix B.

4. EXPERIMENTAL SETUP

This section describes our research questions and how they will be answered empirically.

	 P: ifGt Q: add R: sub S: assi cfg cfg cfg cfg G2
Walks of length 1: $A \to B$, $B \to C$, $A \to C$	Walks of length 1: $P \to Q$, $P \to R$, $Q \to S$, $R \to S$
Walks of length 2: $A \to B \to C$	Walks of length 2: $P \to Q \to S$, $P \to R \to S$

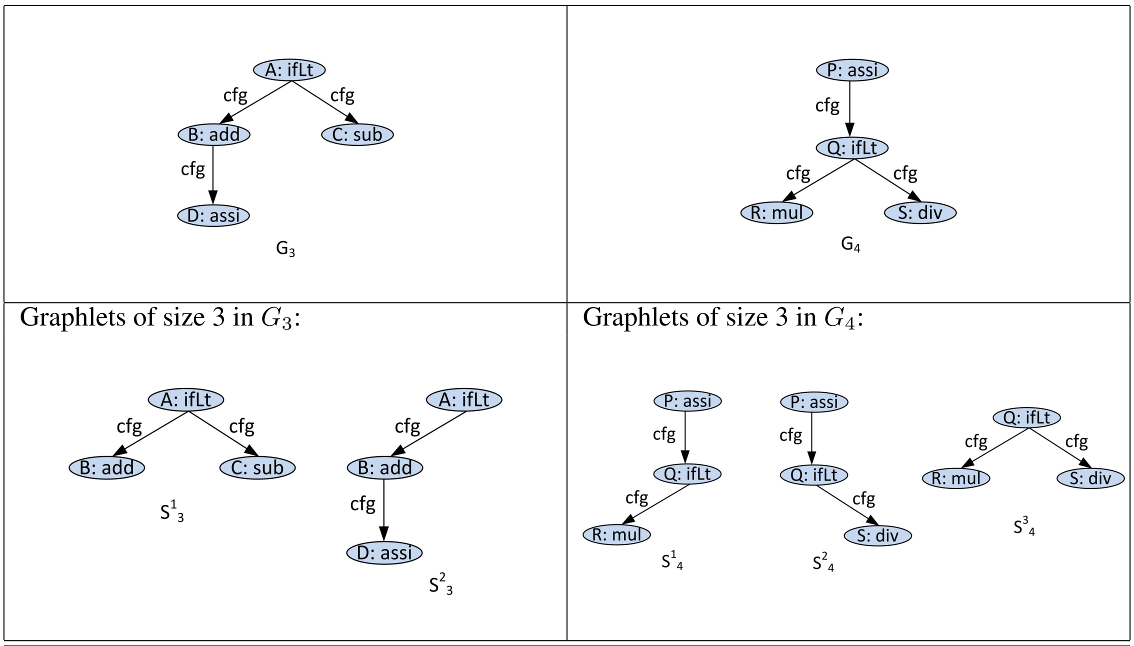
Computation of similarity score between two graphs (restricted to walks up to length 2): $k_{rw}(G_1, G_2) = k_{walk}(A \to B, P \to Q) + k_{walk}(A \to B, P \to R) + ... + k_{walk}(A \to B \to C, P \to Q \to S) + k_{walk}(A \to B \to C, P \to R \to S)$
Computation of similarity score between two walks: $k_{walk}(A \to B, P \to Q) = k_{step}((A,B),(P,Q))$ ... $k_{walk}(A \to B \to C, P \to R \to S) = k_{step}((A,B),(P,R)) \times k_{step}((B,C),(R,S))$
Computation of similarity score between two steps: $k_{step}((A,B),(P,Q)) = k_{node}(A,P) \times k_{node}(B,Q) \times k_{edge}((A,B),(P,Q))$ $k_{step}((A,B),(P,R)) = k_{node}(A,P) \times k_{node}(B,R) \times k_{edge}((A,B),(P,R))$ $k_{step}((B,C),(R,S)) = k_{node}(B,R) \times k_{node}(C,S) \times k_{edge}((A,B),(P,Q))$
Computation of similarity score between two nodes: $k_{node}(A,P) = 0.5$ (two labels have similar properties) $k_{node}(B,Q) = 1$ (two labels are identical) $k_{node}(B,R) = 0$ (two labels are dissimilar) $k_{node}(C,S) = 1$
Computation of similarity score between two edges: $k_{edge}((A,B),(P,Q)) = 1$ (two edges have the same labels) $k_{edge}((A,B),(P,R)) = 1$

Figure 5. Random walk kernel computation for the graphs G_1 and G_2. k_{rw}: kernel value between two graphs. k_{walk}: kernel value between two walks. k_{step}: kernel value between two steps. k_{node}: kernel value between two nodes. k_{edge}: kernel value between two edges.

4.1. Research questions

We conducted experiments to seek answers for the following research questions:

- **RQ1: Are graph kernels more effective in predicting metamorphic relations than node/path features?** In previous work we used a set of features that represent node/path information of control flow graphs for creating predictive models [16]. In this work we extend the feature extraction using graph kernels. We created separate models using the node/path features used in our previous study and the two graph kernels and compared the effectiveness.
- **RQ2: What substructures in graphs are more suitable for predicting metamorphic relations?** This is answered by using two graph kernels that use small subgraphs or all walks in the graphs that represent the functions.
- **RQ3: Is control flow information more effective in predicting metamorphic relations than data dependency information?** The two previous research questions focused on feature



Computation of similarity score between two graphs using graphlets:

$k_{graphlet}(G_1, G_2) = k_{subgraph}(S_3^1, S_4^1) + k_{subgraph}(S_3^1, S_4^2) + k_{subgraph}(S_3^1, S_4^3) + ... + k_{subgraph}(S_3^2, S_4^2) + k_{subgraph}(S_3^2, S_4^3)$

Computation of similarity score between two graphlets:

$k_{subgraph}(S_3^1, S_4^1) = 0$
$k_{subgraph}(S_3^1, S_4^2) = 0$
$k_{subgraph}(S_3^1, S_4^3) = k_{node}(A, Q) \times k_{node}(B, R) \times k_{node}(C, S) \times k_{edge}((A, B), (Q, R)) \times k_{edge}((A, C), (Q, S))$
...
$k_{subgraph}(S_3^2, S_4^2) = k_{node}(A, P) \times k_{node}(B, Q) \times k_{node}(D, S) \times k_{edge}((A, B), (P, Q)) \times k_{edge}((B, D), (Q, S))$
$k_{subgraph}(S_3^2, S_4^3) = 0$

Computation of similarity score between two nodes:

$k_{node}(A, Q) = 1$
$k_{node}(B, R) = 0.5$
$k_{node}(C, S) = 0.5$
...
$k_{node}(D, S) = 0$

Computation of similarity score between two edges:

$k_{edge}((A, B), (Q, R)) = 1$
...
$k_{edge}((B, D), (Q, S)) = 1$

Figure 6. Graphlet kernel computation for graphs G_3 and G_3. $k_{graphlet}$: graphlet kernel value. $k_{subgraph}$: kernel value of between two subgraphs. k_{node}: kernel value between two nodes. k_{edge}: kernel value between two steps.

extraction. Prediction accuracy can also depend on the source of program information used to create the model. In this work we use two types of information about the program: control flow information and data dependency information. We compare the effectiveness of these two types of information for predicting metamorphic relations.

Table I. The metamorphic relations used in this study.

Relation	Change made to the input	Expected change in the output
Permutative	Randomly permute the elements	Remain constant
Additive	Add a positive constant	Increase or remain constant
Multiplicative	Multiply by a positive constant	Increase or remain constant
Invertive	Take the inverse of each element	Decrease or remain constant
Inclusive	Add a new element	Increase or remain constant
Exclusive	Remove an element	Decrease or remain constant

- **RQ4: Will combining control flow information with data dependency information improve the prediction effectiveness of metamorphic relations?** We further look at whether creating prediction models that use both control flow and data dependency information can improve prediction effectiveness.

4.2. *The code corpus*

To measure the effectiveness of our proposed methods, we built a code corpus containing 100 functions that take numerical inputs and produce numerical outputs. We extended the corpus used in our previous study [16] with functions from the following open source projects:

1. The Colt Project[‡]: set of open source libraries written for high performance scientific and technical computing in Java
2. The Apache Mahout[§]: a machine learning library written in Java
3. The Apache Commons Mathematics Library[¶]: library of mathematics and statistics components written in the Java

We list these functions in Table III in Appendix D[‖].

4.3. *Metamorphic relations*

We used the six metamorphic relations shown in Table I. These metamorphic relations were identified by Murphy et al. [25] and are commonly found in mathematical functions. We list the input modifications and the expected output modification of these metamorphic relations in Table I. A function f is said to satisfy (or exhibit) a metamorphic relation m in Table I, if the change in the output is according to what is expected after modifying the original input. Previous studies have shown that these are the type of metamorphic relations that tend to be identified by humans [26, 27, 28]. In this work we use the term *positive instance* to refer to a function that satisfies a given metamorphic relation and the term *negative instance* to refer to a function that does not satisfy a considered metamorphic relation. Table II reports the number of positive and negative instances for each metamorphic relation.

[‡]http://acs.lbl.gov/software/colt/
[§]https://mahout.apache.org/
[¶]http://commons.apache.org/proper/commons-math/
[‖]These fuctions and their graph representations can be accessed via the following URL: `http://www.cs.colostate.edu/~upuleegk/data/functions.tar.gz`

Table II. Number of positive and negative instances for each metamorphic relation.

Metamorphic Relation	**#Positive**	**#Negative**
Permutative	34	66
Additive	57	43
Multiplicative	68	32
Invertive	65	35
Inclusive	33	67
Exclusive	31	69

4.4. Evaluation procedure

We used 10-fold stratified cross validation in our experiments. In 10-fold cross validation the data set is randomly partitioned into 10 subsets. Then nine subsets are used to build the predictive model (training) and the remaining subset is used to evaluate the performance of the predictive model (testing). This process is repeated 10 times in which each of the 10 subsets is used to evaluate the performance. In stratified 10-fold cross validation, 10 folds are partitioned in such away that the folds contain approximately the same proportion of positive instances (functions that exhibit a specific metamorphic relation) and negative instances (functions that do not exhibit a specific metamorphic relation) as in the original data set. Results are generated by averaging over 10 runs of cross validation.

We used nested cross validation to select the regularization parameter (C) of the SVM as well as the parameters of the graph kernels. In nested cross validation, values for parameters are selected by performing cross validation on training examples of each fold. We used the SVM implementation in the PyML Toolkit** in this work.

We used two evaluation measures when reporting our results. The first evaluation measure is the *balanced success rate (BSR)* [29]. Standard accuracy gives the fraction of the correctly classified instances in the data set. Therefore it is not a good measure to evaluate success when the data set is unbalanced as in our case. BSR considers the imbalance in data as follows:

$$BSR = \frac{P(success|+)+P(success|-)}{2},$$

where $P(success|+)$ is the estimated probability of classifying a positive instance correctly and $P(success|-)$ is the probability of classifying a negative instance correctly.

The second evaluation measure is the area under the receiver operating characteristic curve (AUC) value. 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 [30]. Therefore 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 that classifies randomly will have AUC = 0.5. Previous studies have shown that AUC is a better measure for comparing learning algorithms [30]. Since the AUC does not depend on the discrimination threshold of the classifier we use the AUC measure for comparisons.

**http://pyml.sourceforge.net/

5. RESULTS

We present the results of our empirical studies in this section. We first evaluated the effectiveness of the two graph kernels used in this study. We specifically looked at how the effectiveness of these kernels vary with the values given for the parameters in these two kernels. Then we compared the effectiveness of the graph kernels with the features used in our previous work. Finally we compared the effectiveness of control flow and data dependency information for predicting MRs.

5.1. *Effectiveness of the random walk kernel*

With the random walk kernel, we first evaluated how the prediction accuracy changes with the λ parameter in the kernel. For this evaluation we computed the random walk kernel values using only the control flow edges in the graph. We varied the value of λ from 0.1 to 0.9 and evaluated the performance using 10 fold stratified cross validation. We used the kernel normalization described in Appendix C to normalize the random walk kernel. We selected the regularization parameter of the SVM using nested cross validation. Figure 7 shows the variation of BSR and AUC with λ for each MR. For some MRs, such as additive and multiplicative MRs, there was a considerable variation in accuracy with λ. For these two MRs, higher values of λ gave better performance than lower values of λ. Since each walk of length n is weighted by λ^n, with higher λ values, the random walk kernel value will have a higher contribution from longer walks (see Equation (8) in Appendix A). Therefore, for predicting additive and multiplicative MRs, the contribution from long walks in the CFGs seems to important. For the other four MRs the accuracy did not vary considerably with λ. These results show that the length of random walks has an impact on the prediction effectiveness of some MRs.

Next, we evaluated the effects of the modifications that we made to the node kernel used with the random walk kernel. For this we created separate prediction models using k_{node^1} and k_{node^2} described in equation 4 and equation 5 in Appendix A. We used nested cross validation to select the regularization parameter of the SVM and the λ parameter of the random walk kernel. Figure 8 shows the performance comparison between k_{node^1} and k_{node^2}. Figure 8b shows that the modified node kernel k_{node^2} improves the prediction effectiveness of all the metamorphic relations. Therefore in what follows we use k_{node^2}.

To identify whether adding more functions to our code corpus would help to increase the accuracy, we plotted leaning curves for the random walk kernel. A learning curve shows how the prediction accuracy varies with the training set size. Figure 9 shows the learning curves for the six MRs. To generate these curves we held 10% of the functions in our code corpus as the test set. From the other functions we selected subsets 10% to 100% as the training set. We repeated this process 10 times so that the test set would have a different set of functions each time. For all the MRs the AUC values incresed as the training set size increses. But the AUC values did not converge indicating that adding more training instances might improve the prediction accuracy further.

5.2. *Effectiveness of the graphlet kernel*

With the graphlet kernel, we first evaluated how prediction accuracy varies with graphlet size. For this evaluation, we used only the control flow edges in the graphs when computing the graphlet

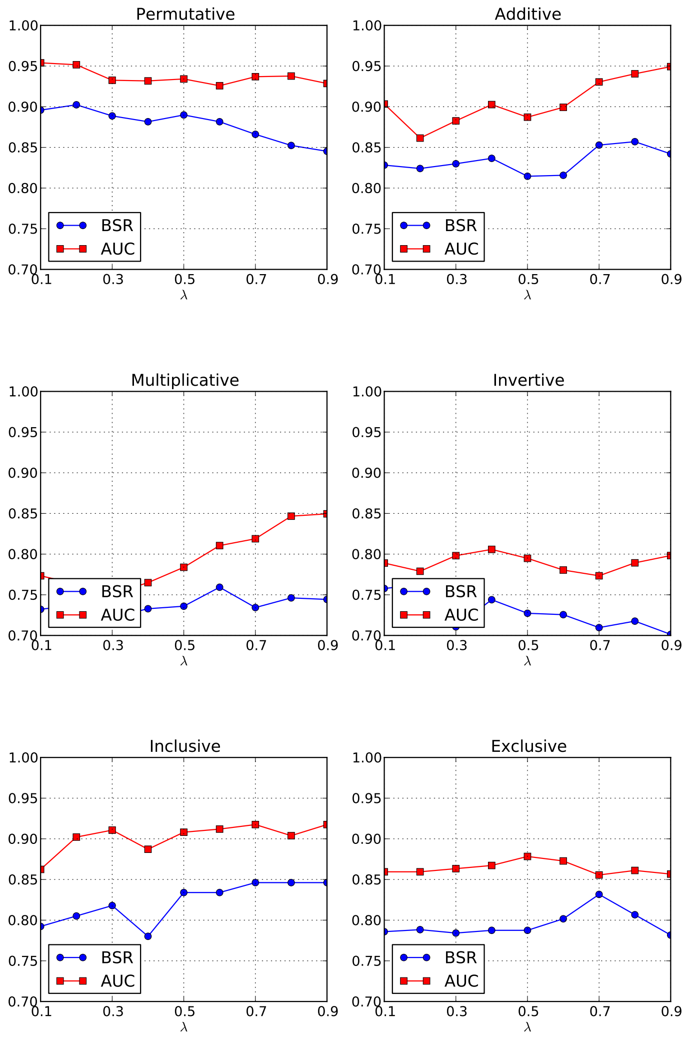
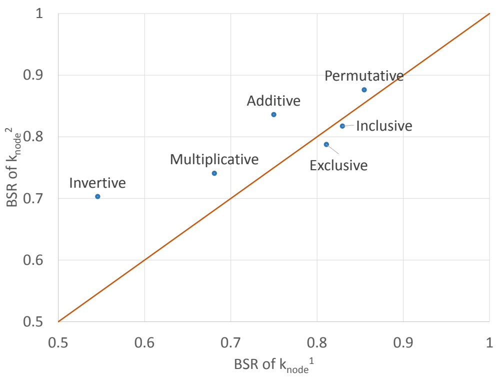
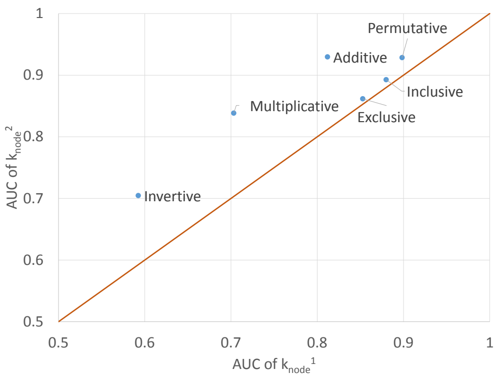


Figure 7. Variation of the BSR and the AUC with the parameter λ in the random walk graph kernel for each MR.

kernel values. We used cosine normalization described in Appendix C to normalize the random walk kernel. We varied the graphlet size from three nodes to five nodes and created separate predictive models. In addition, we used all the graphlets together and created a single predictive model. Figure 10a and Figure 10b shows how the average BSR and average AUC varies with the graphlet size for each MR. When predicting the multiplicative MR, predictive model created using graphlets of size 5 performed the best. For the invertive MR, graphlets with three nodes performed better than the other predictive models. For the other MRs, all the predictive models gave a similar performance.



(a) Comparison of BSR



(b) Comparison of AUC

Figure 8. Performance comparison of k_{node^1} and k_{node^2} for the random walk kernel.

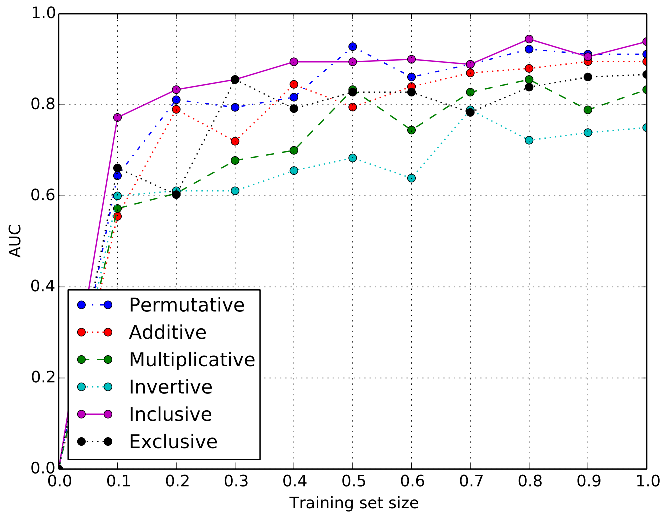


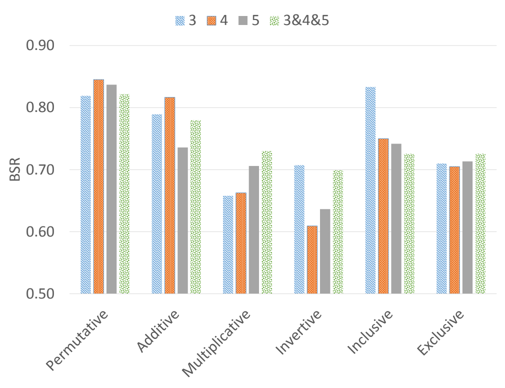
Figure 9. Learning curves for the six metamorphic relations

Similar to the random walk kernel, we also compared the performance of the graphlet kernel with the node kernels k_{node^1} and k_{node^2}. As with the random walk kernel, the graphlet kernel with k_{node^2} performed better than the graphlet kernel with k_{node^1}.

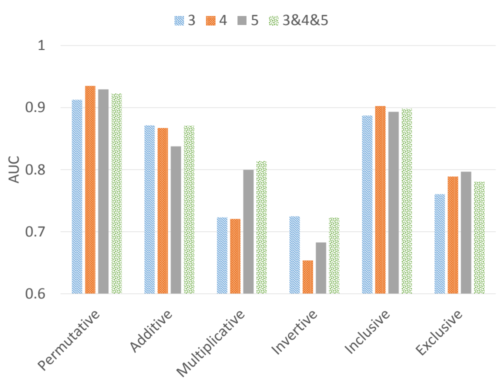
5.3. *Comparison of feature extraction methods*

We compared the performance of the node/path features that we used in our initial study [16] with the two graph kernels used in this study. Below we present the details of these three feature extraction methods used for this evaluation:

1. Node/path features: we followed the same protocol as our earlier work [16]. Node/path features were calculated using only the "cfg" edges of the graphs. Node features are created by combining the operation performed in the node, its in-degree and its out-degree. Path features are created by taking the sequence of nodes in the shortest path from N_{start} to each node and



(a) BSR



(b) AUC

Figure 10. Variation of performance with the graphlet size. 3, 4 and 5 are the predictive models created using graphlets of size 3, 4 and 5 respectively. 3&4&5 is the performance of the predictive model created using all the graphlets.

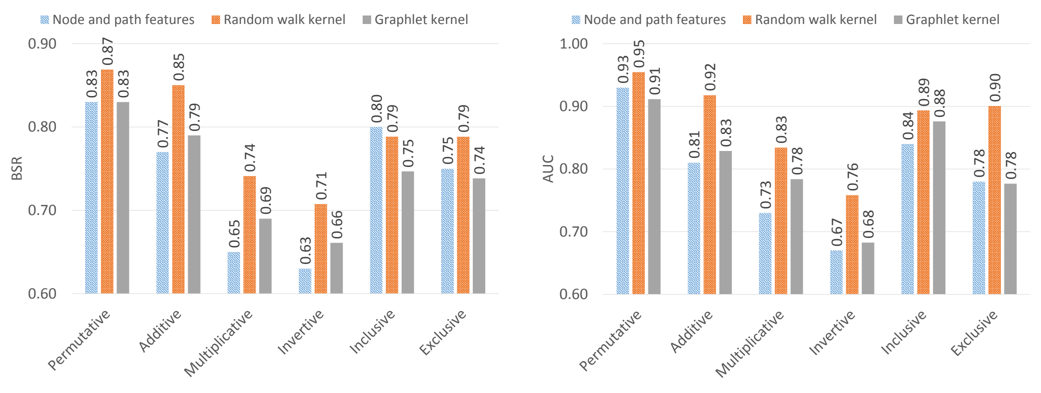
the sequence of nodes in the shortest path from each node N_{exit}. We used the linear kernel over these features as it exhibited the best performance in our previous experiments [16].

2. Random walk kernel: we computed the random walk kernel using only the “cfg” edges of the graphs.
3. Graphlet kernel: we computed the graphlet kernel using only the “cfg” edges of the graphs.

Figure 11a and Figure 11b shows the average BSR and average AUC for the three feature extraction approaches: node and path features, random walk kernel and graphlet kernel. Among the three feature extraction approaches, the random walk kernel gave the best prediction accuracy for all the MRs. Except for the permutative MR, the AUC values of the graphlet kernel were equal or greater than the AUC values of node/path features. This improvement in performance could be attributed to the flexibility provided by the graph kernels. The node kernel can be used to compare high level properties of the operations such as commutativity of mathematical operations, in addition to direct comparison of node labels. In fact, figure 8 shows that using such high level properties of operations would improve the accuracy of the prediction models.

Our results show that the random walk kernel performs better than the graphlet kernel for most of the MRs. The random walk kernel compares two control flow graphs based on their walks. A walk in the control flow graph corresponds to a potential execution trace of the function. Therefore it computes a similarity score between two programs based on potential execution traces. The graphlet kernel compares two control flow graphs based on small subgraphs. These subgraphs will capture decision points in the program such as if-conditions. A metamorphic relation is a relationship between outputs produced by multiple executions of the function. Some execution traces directly correlate with some metamorphic relations. Therefore the random walk kernel, which uses execution traces to compares two functions should perform better than the graphlet kernel.

We also evaluated the effect of adding the random walk kernel and the graphlet kernel. Adding the two kernel values is equivalent to using both graph substructures, walks and subgraphs for creating a single model. We added the random walk kernel value and the graphlet kernel value computed for a pair of functions and created a single kernel matrix with these added values. Our experiments showed that this combined kernel could not outperform the random walk kernel. Therefore we do not discuss these results in detail.



(a) BSR (b) AUC

Figure 11. Prediction accuracy of node and path features, random walk kernel and graphlet kernel.

5.4. *Effectiveness of control flow and data dependency information*

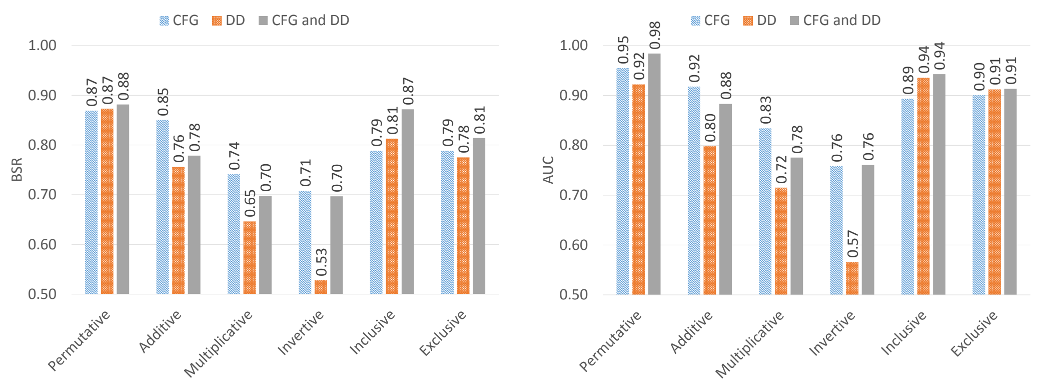
Finally, we compared the effectiveness of the control flow and data dependency information of a function for predicting metamorphic relations. Figure 12 shows the effectiveness of using only control flow information, only data dependency information and both control flow and data dependency information. Since the random walk kernel performed best, we used it for this analysis. For this experiment we computed the random walk kernel values separately using the following edges in the graphs: (1) only "cfg" edges, (2) only "dd" edges and (3) both "cfg" and "dd" edges.

We observe that overall, "cfg" edges performed much better than "dd" edges. Performance using "dd" edges was particularly low for the invertive MR. This can be explained by the fact that the control flow graph directly represents information about the execution of the program compared to the data dependency information. Typically, combining informative features improves classifier performance. We observed this effect in four out of the six MRs. The reduced performance for the multiplicative MR might be the result of the poor performance of the "dd" edges.

6. THREATS TO VALIDITY

External validity: We used a code corpus consisting of 100 mathematical functions for our empirical study. These functions differ in size and complexity. They also perform different functionalities. But this set of functions can pose a threat to external validity. We tried to minimize this threat by using functions obtained from different open source libraries.

Internal validity: Threats to internal validity can occur due to potential faults in the implementations of the functions. Since we were not the developers of these functions we cannot guarantee that these functions are free of faults. The competent programmer hypothesis [31] states that 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. We use control



(a) BSR (b) AUC

Figure 12. Performance of control flow and data dependency information using random walk graph kernel. CFG - using only CFG edges, DD - using only data dependency edges, CFG and DD - using both CFG and data dependency edges.

flow and data dependency information about a program to create our prediction models. According to the competent programmer hypothesis these information should not change significantly even with a fault. In addition, there may be more relevant metamorphic relations for these functions than the six metamorphic relations that we used for our empirical studies.

Construct validity: Third party tools that we used in this work can pose threats to construct validity. We used the Soot framework to create the CFGs for the functions and annotate them with data dependency information. Further we used the NetworkX[††] package for graph manipulation. To minimize these threats we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Conclusion validity: We used AUC value for evaluating the performance of the classifiers. We considered $AUC \geqslant 0.80$ as a good classifier. This is cosistent with most of the machine learning literature.

7. RELATED WORK

Metamorphic testing has been used to test applications without oracles in different areas. Xie et al. [14] used metamorphic testing to test machine learning applications. Metamorphic testing was used to test simulation software such as health care simulations [32] and Monte Carlo modeling [33]. Metamorphic testing has been used effectively in bioinformatics [34], computer graphics [35] and for testing programs with partial differential equations [36]. Murphy et al. [37] 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. However they specify the metamorphic relations manually.

Metamorphic testing has also been used to test programs at the system level. Murphy et al. developed a method for automating system level metamorphic testing [38]. In this work, they

[††]https://networkx.github.io/

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.

Our previous work showed that machine learning methods can be used to predict metamorphic relations in previously unseen functions [16]. We used features obtained from control flow graphs to train our prediction models. We showed that when predicting metamorphic relations, support vector machines achieved cross validation accuracies ranging from 83%-89% depending on the metamorphic relation.

Machine learning techniques 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. [39]. Bowring et al. used program execution data to classify program behavior [40]. Briand et al. [41] used the C4.5 machine learning algorithm for test suite refinement. Briand et al. [42] 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 determine if a failure occurred due to the same fault(s).

Frounchi et al. [43] 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 [44] used a SVM model for predicting software reliability. Wang et al. [45] used SVMs for creating test oracles for reactive systems. They first collect test traces from the program under test and obtain pass/fail decisions from an alternate method such as a domain expert. Then they train a SVM classification model using these labeled test traces. The trained SVM classifier work as the oracle for classifying previously unseen test traces. We did not find any applictions of graph kernels in software engineering.

8. CONCLUSIONS AND FUTURE WORK

Metamorphic testing is a useful technique for testing programs for which an oracle is not available. Identifying a set of metamorphic relations that should be satisfied by the program is an important initial task that determines the effectiveness of metamorphic testing, which is currently done manually. Our work investigates novel machine learning based approaches for predicting metamorphic relations for a given function automatically. Our previous work showed that classification models created using a set of features extracted from control flow graphs are highly effective in predicting metamorphic relations. Further, we showed that these prediction models make consistent predictions even when the functions contain faults. In this work we extended our previous work in two directions. First, we extend the feature extraction using graph kernels, which is a more systematic way to extract features from graphs. Secondly, we used data dependency information of a function in addition to the control flow information used in our previous work. We performed a series of empirical studies using a set of functions obtained from open source code libraries. The results of our empirical studies show that graph kernels improve the prediction effectiveness of metamorphic relations. But it is important to select the proper graph substructure when selecting the graph kernels. Random walk kernel seems to be the most effective graph kernel for predicting most metamorphic relations. In addition, incorporating high level information about the operations such as properties of the mathematical operations in to the kernel further improved prediction accuracy.

When considering the control flow and data dependency information, control flow information of a function seems to be the most effective for predicting metamorphic relations. But, for some metamorphic relations using both types of information together improves the accuracy. Therefore selecting what program properties to use for computing the graph kernel valued could be done using nested cross validation on the training set in addition to selecting parameters of the graph kernels.

As future work, we plan to develop multi-label classifiers that can predict several metamorphic relations using a single classifier. A multi-label classifier can predict several labels associated with a data instance at once [46]. A given function could satisfy multiple metamorphic relations and there can be dependencies among these metamorphic relations. Such relationships can be utilized by multi-label classifiers. In addition we plan to incorporate dynamic properties of programs such as dynamic execution traces for predicting metamorphic relations.

A. DEFINITION OF THE RANDOM WALK KERNEL

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graph representations of programs as described in Section 3.1. Consider two walks, $walk_1$ in G_1 and $walk_2$ in G_2. $walk_1 = (v_1^1, v_1^2, ..., v_1^{n-1}, v_1^n)$ where $v_1^i \in V_1$ for $1 \leq i \leq n$. $walk_2 = (v_2^1, v_2^2, ..., v_2^{n-1}, v_2^n)$ where $v_2^i \in V_2$ for $1 \leq i \leq n$. $(v_1^i, v_1^{i+1}) \in E_1$ and $(v_2^i, v_2^{i+1}) \in E_2$. Then the kernel value of two graphs can be defined as

$$k_{rw}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2), \tag{1}$$

where the walk kernel k_{walk} can be defined as

$$k_{walk}(walk_1, walk_2) = \prod_{i=1}^{n-1} k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})). \tag{2}$$

The kernel for each step will be defined using the kernel values of the two node pairs and the edge pair of the considered step as follows:

$$k_{step}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = k_{node}(v_1^i, v_2^i) * k_{node}(v_1^{i+1}, v_2^{i+1}) * k_{edge}((v_1^i, v_1^{i+1}), (v_2 i, v_2^{i+1})). \tag{3}$$

We defined two node kernels: k_{node^1} and k_{node^2} to get the similarity score between two nodes. We use k_{node^1} and k_{node^2} in place of k_{node} in equation (3). k_{node^1} checks the similarity of the node labels.

$$k_{node^1}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0 & otherwise. \end{cases} \tag{4}$$

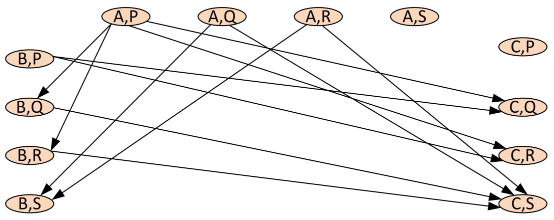


Figure 13. Direct product graph of G_1 and G_1 in Table 5

In the second node kernel, k_{node^2} we considered whether the operation performed in the two nodes are in the same group if the node labels are not equal. For this study we grouped the mathematical operations using commutative and associative properties.

$$k_{node^2}(v_i, v_j) = \begin{cases} 1 & \text{if } label(v_i) = label(v_j), \\ 0.5 & \text{if } group(v_i) = group(v_j) \text{ and } label(v_i) \neq label(v_j), \\ 0 & \text{if } group(v_i) \neq group(v_j) \text{ and } label(v_i) \neq label(v_j). \end{cases} \quad (5)$$

The edge kernel, k_{edge} is defined as follows:

$$k_{edge}((v_1^i, v_1^{i+1}), (v_2^i, v_2^{i+1})) = \begin{cases} 1 & \text{if } label(v_1^i, v_1^{i+1}) = label(v_2^i, v_2^{i+1}), \\ 0 & otherwise. \end{cases} \quad (6)$$

We used the *direct product graph* approach presented by Gärtner et al. [20] with the modification introduced by Borgwardt et al. [47] for calculating all the walks within two graphs. The direct product graph of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is denoted by $G_1 \times G_2$. The nodes and edges of the direct product graph are defined as follows:

$$\begin{aligned} V_X(G_1 \times G_2) &= \{(v_1, v_2) \in V_1 \times V_2\} \\ E_X(G_1 \times G_2) &= \{((v_1^1, v_1^2), (v_2^1, v_2^2)) \in V^2(G_1 \times G_2) : (v_1^1, v_1^2) \in E_1 \wedge (v_2^1, v_2^2) \in E_2 \wedge \\ &\qquad label(v_1^1, v_1^2) = label(v_2^1, v_2^2)\} \end{aligned}$$

Figure 13 shows the direct product graph of the two graphs G_1 and G_2 in Table 5. As shown in figure 13, the direct product graph has a node for each pair of nodes in G_1 and G_2. There is an edge between two nodes in the product graph if there are edges between the two corresponding pairs of the nodes in G_1 and G_2. Taking a walk in the direct product graph is equivalent to taking simultaneous walks in G_1 and G_2. Consider the walk $AP \rightarrow BQ \rightarrow CS$ in the direct product graph. This walk represents taking the walks $A \rightarrow B \rightarrow C$ in G_1 and $P \rightarrow Q \rightarrow S$ in G_2 simultaneously. Therefore by modifying the adjacency matrix of the direct product graph to contain similarity scores between steps, instead of having one/zero values, we can use the adjacency matrix of the direct product graph to efficiently compute the random walk kernel value of two graphs. We present the definition of the direct product graph and how it is used to compute the random walk kernel in Section A.

Based on the product graph, the random walk kernel is defined as:

$$k_{rw}(G_1, G_2) = \sum_{i,j=1}^{V_X} \left[\sum_{n=0}^{\infty} \lambda^n A_X^n \right]_{ij} \tag{8}$$

where A_X denotes the adjacency matrix of the direct product graph and $1 > \lambda \geq 0$ is a weighting factor. To make the sum finite, we limited limited n in Equation (8) to 10 in our experiments. The adjacency matrix of the product graph is modified as follows to include k_{step} defined in Equation (3):

$$[A_X]_{((v_i,w_i),(v_j,w_j))} = \begin{cases} k_{step}((v_i, w_i), (v_j, w_j)) & \text{if } ((v_i, w_i), (v_j, w_j)) \in E_X \\ 0 & \text{otherwise} \end{cases}$$

B. DEFINITION OF THE GRAPHLET KERNEL

Shervashidze et al. [18] developed a graph kernel that compares all subgraphs with $k \in \{3, 4, 5\}$ nodes. The authors refer to this kernel as the *graphlet kernel*, which was developed for unlabeled graphs. We extended the graphlet kernel for directed labeled graphs since the labels in our graph based program represent important semantic information about the function. We first describe the original graphlet kernel and then describe the modifications we made to it.

Let a graph be a pair $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ are the n vertices and $E \subseteq V \times V$ is the set of edges. Given $G = (V, E)$ and $H = (V_H, E_H)$, H is said to be a subgraph of G iff there is an injective mapping $\alpha : V_H \to V$ such that $(v, w) \in E_H$ iff $(\alpha(v), \alpha(w)) \in E$. If H is a subgraph of G it is denoted by $H \sqsubseteq G$.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a bijective mapping $g : V_1 \to V_2$ such that $(v_i, v_j) \in E_1$ iff $(g(v_i), g(v_j)) \in E_2$. If G_1 and G_2 are isomorphic, it is denoted by $G_1 \simeq G_2$ and g is called the isomorphism function.

Let $\mathcal{M}_1^k$ and $\mathcal{M}_2^k$ be the set of size k subgraphs of the graphs G_1 and G_2 respectively. Let $S_1 = (V_{S_1}, E_{S_1}) \in \mathcal{M}_1^k$ and $S_2 = (V_{S_2}, E_{S_2}) \in \mathcal{M}_2$. Then the graphlet kernel, $k_{graphlet}(G_1, G_2)$ is computed as

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} \delta(S \simeq S_2) \tag{9}$$

where

$$\delta(S_1 \simeq S_2) = \begin{cases} 1 & \text{if } S_1 \simeq S_2 \\ 0 & otherwise \end{cases}$$

The kernel in Equation (9) is developed for unlabeled graphs. To consider the node labels and edge labels we modified the kernel in 9 as follows:

$$k_{graphlet}(G_1, G_2) = \sum_{S_1 \in \mathcal{M}_1^k} \sum_{S_2 \in \mathcal{M}_2^k} k_{subgraph}(S_1, S_2) \tag{10}$$

where

$$k_{subgraph}(S_1, S_2) = \begin{cases} \prod_{v \in V_{S_1}} k_{node}(v, g(v)) * \prod_{(v_i, v_j) \in E_{S_1}} k_{edge}((v_i, v_j), (g(v_i), g(v_j))) & S_1 \simeq S_2, \\ 0 & otherwise. \end{cases} \tag{11}$$

We used Equation (10) to compute the graphlet kernel value for a pair of programs represented in the graph based representation described in Section 3.1. Similar to the random walk kernel, k_{node} in Equation (11) is replaced by k_{node^1} and k_{node^2} defined in Equation (4) and Equation(5) respectively. Equation (6) defines k_{edge}.

C. KERNEL NORMALIZATION

We normalize each kernel such that each example has a unit norm by the exprssion [29, 17]:

$$k'_{graph}(G_1, G_2) = \frac{k_{graph}(G_1, G_2)}{\sqrt{k_{graph}(G_1, G_1) k_{graph}(G_2, G_2)}} \tag{12}$$

D. LIST OF FUNCTIONS

Table III lists the functions used in our experiments.

ACKNOWLEDGEMENT

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.

REFERENCES

1. Sanders R, Kelly D. The challenge of testing scientific software. *Proceedings Conference for the Association for Software Testing (CAST)*, Toronto, 2008; 30–36.
2. Sanders R, Kelly D. Dealing with risk in scientific software development. *Software, IEEE* july-aug 2008; **25**(4):21 –28, doi:10.1109/MS.2008.84.
3. Hatton L. The t experiments: errors in scientific software. *Computational Science Engineering, IEEE* apr-jun 1997; **4**(2):27 –38, doi:10.1109/99.609829.
4. Abackerli AJ, Pereira PH, Calônego Jr N. A case study on testing CMM uncertainty simulation software (VCMM). *Journal of the Brazilian Society of Mechanical Sciences and Engineering* 03 2010; **32**:8 – 14.

Table III. Functions used in the experiment.

Open source project	**Functions used in the experiment**
Colt Project	min, max, covariance, durbinWatson, lag1, meanDeviation, product, weightedMean, autoCorrelation, binarySearchFromTo, quantile, sumOfLogarithms, kurtosis, pooledMean, sampleKurtosis, sampleSkew, sampleVariance, pooledVariance, sampleWeightedVariance, skew, standardize, weightedRMS, harmonicMean, sumOfPowerOfDeviations, power, square, winsorizedMean, polevl
Apache Mahout	add, cosineDistance, manhattanDistance, chebyshevDistance, tanimotoDistance, hammingDistance, sum, dec, errorRate
Apache Commons Mathematics Library	errorRate, scale, eucleadianDistance, distance1, distanceInf, ebeAdd, ebeDivide, ebeMultiply, ebeSubtract, safeNorm, entropy, g, calculateAbsoluteDifferences, calculateDifferences, computeDividedDifference, computeCanberraDistance, evaluateHoners, evaluateInternal, evaluateNewton, mean, meanDifference, variance, varianceDifference, equals, checkNonNegative, checkPositive, chiSquare, evaluateWeightedProduct, partition, geometricMean, weightedMean, median, dotProduct
Functions from the previous study [16]	reverse, add_values, bubble_sort, shell_sort, sequential_search, selection_sort, array_calc1, set_min_val, get_array_value, find_diff, array_copy, find_magnitude, dec_array, find_max2, insertion_sort, mean_absolute_error, check_equal_tolerance, check_equal, count_k, clip, elementwise_max, elementwise_min, count_non_zeroes, cnt_zeroes, elementwise_equal, elementwise_not_equal, select

5. Miller G. A scientist's nightmare: Software problem leads to five retractions. *Science* 2006; **314**(5807):1856–1857, doi:10.1126/science.314.5807.1856. URL http://www.sciencemag.org/content/314/5807/1856.short.
6. Hatton L, Roberts A. How accurate is scientific software? *Software Engineering, IEEE Transactions on* oct 1994; **20**(10):785 –797, doi:10.1109/32.328993.
7. Dubois P. Testing scientific programs. *Computing in Science Engineering* july-aug 2012; **14**(4):69 –73, doi: 10.1109/MCSE.2012.84.
8. Weyuker EJ. On testing non-testable programs. *Comput. J.* 1982; **25**(4):465–470.
9. Joppa LN, McInerny G, Harper R, Salido L, Takeda K, O'Hara K, Gavaghan D, Emmott S. Troubling trends in scientific software use. *Science* 2013; **340**(6134):814–815, doi:10.1126/science.1231535. URL http://www.sciencemag.org/content/340/6134/814.short.
10. Kanewala U, Bieman JM. Testing scientific software: A systematic literature review. *Information and Software Technology* 2014; **56**(10):1219 – 1232, doi:http://dx.doi.org/10.1016/j.infsof.2014.05.006. URL http://www.sciencedirect.com/science/article/pii/S0950584914001232.
11. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01*, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong 1998.
12. Chen TY, Tse TH, Zhou ZQ. Fault-based testing without the need of oracles. *Information and Software Technology* 2003; **45**(1):1–9.
13. Zhou ZQ, Huang DH, Tse TH, Yang Z, Huang H, Chen TY. Metamorphic testing and its applications. *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, Software Engineers Association, 2004.
14. Xie X, Ho JW, Murphy C, Kaiser G, Xu B, Chen TY. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 2011; **84**(4):544 – 558, doi:10.1016/j.jss.2010.11.920.
15. Chen TY, Huang DH, Tse TH, Zhou ZQ. Case studies on the selection of useful relations in metamorphic testing. *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, 2004; 569–583.
16. Kanewala U, Bieman J. Using machine learning techniques to detect metamorphic relations for programs without test oracles. *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013; 1–10,

doi:10.1109/ISSRE.2013.6698899.

17. Shawe-Taylor J, Cristianini N. *Kernel Methods for Pattern Analysis*. Cambridge University Press: New York, NY, USA, 2004.
18. Shervashidze N, Vishwanathan SVN, Petri T, Mehlhorn K, Borgwardt K. Efficient graphlet kernels for large graph comparison. *Proceedings of the International Workshop on Artificial Intelligence and Statistics. Society for Artificial Intelligence and Statistics*, 2009.
19. Borgwardt K, Kriegel HP. Shortest-path kernels on graphs. *Data Mining, Fifth IEEE International Conference on*, 2005; 8 pp.–, doi:10.1109/ICDM.2005.132.
20. Gärtner T, Flach P, Wrobel S. On graph kernels: Hardness results and efficient alternatives. *Learning Theory and Kernel Machines*, *Lecture Notes in Computer Science*, vol. 2777, Schölkopf B, Warmuth M (eds.). Springer Berlin Heidelberg, 2003; 129–143, doi:10.1007/978-3-540-45167-9_11.
21. Ramon J, Grtner T. Expressivity versus efficiency of graph kernels. *Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences*, 2003; 65–74.
22. Allen FE. Control flow analysis. *SIGPLAN Not.* Jul 1970; **5**(7):1–19, doi:10.1145/390013.808479. URL http://doi.acm.org/10.1145/390013.808479.
23. Vallee-Rai R, Hendren LJ. Jimple: Simplifying Java bytecode for analyses and transformations 1998.
24. Kondor RI, Lafferty J. Diffusion kernels on graphs and other discrete structures. *In Proceedings of the ICML*, 2002; 315–322.
25. Murphy C, Kaiser GE, Hu L, Wu L. Properties of machine learning applications for use in metamorphic testing. *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA*, 2008; 867–872.
26. Hu P, Zhang Z, Chan WK, Tse TH. An empirical comparison between direct and indirect test result checking approaches. *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06, ACM: New York, NY, USA, 2006; 6–13, doi:10.1145/1188895.1188901. URL http://doi.acm.org/10.1145/1188895.1188901.
27. Liu H, Kuo FC, Towey D, Chen TY. How effectively does metamorphic testing alleviate the oracle problem? *Software Engineering, IEEE Transactions on* Jan 2014; **40**(1):4–22, doi:10.1109/TSE.2013.46.
28. Mishra G Kunal Swaroop Kaiser. Effectiveness of teaching metamorphic testing. *Technical Report CUCS-020-12*, Department of Computer Science, Columbia University 2012.
29. Ben-Hur A, Weston J. A User's Guide to Support Vector Machines. *Data Mining Techniques for the Life Sciences*, *Methods in Molecular Biology*, vol. 609, Carugo O, Eisenhaber F (eds.). chap. 13, Humana Press: Totowa, NJ, 2010; 223–239, doi:10.1007/978-1-60327-241-4_13. URL http://dx.doi.org/10.1007/978-1-60327-241-4_13.
30. Huang J, Ling C. Using AUC and accuracy in evaluating learning algorithms. *Knowledge and Data Engineering, IEEE Transactions on* march 2005; **17**(3):299 – 310, doi:10.1109/TKDE.2005.50.
31. DeMillo R, Lipton R, Sayward F. Hints on test data selection: Help for the practicing programmer. *Computer* april 1978; **11**(4):34 –41, doi:10.1109/C-M.1978.218136.
32. Murphy C, Raunak MS, King A, Chen S, Imbriano C, Kaiser G, Lee I, Sokolsky O, Clarke L, Osterweil L. On effective testing of health care simulation software. *Proceedings of the 3rd Workshop on Software Engineering in Health Care*, SEHC '11, ACM: New York, NY, USA, 2011; 40–47, doi:10.1145/1987993.1988003.
33. Ding J, Wu T, Wu D, Lu JQ, Hu XH. Metamorphic testing of a Monte Carlo modeling program. *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, ACM: New York, NY, USA, 2011; 1–7, doi:10.1145/1982595.1982597.
34. Chen TY, Ho JWK, Liu H, Xie X. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 2009; **10**.
35. Guderlei R, Mayer J. Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing. *Quality Software, 2007. QSIC '07. Seventh International Conference on*, 2007; 404 –409, doi:10.1109/QSIC.2007.4385527.
36. Chen TY, Feng J, Tse TH. Metamorphic testing of programs on partial differential equations: A case study. *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, IEEE Computer Society: Washington, DC, USA, 2002; 327–333.
37. Murphy C, Shen K, Kaiser G. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, IEEE Computer Society: Washington, DC, USA, 2009; 436–445, doi:10.1109/ICST.2009.19.

38. Murphy C, Shen K, Kaiser G. Automatic system testing of programs without test oracles. *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, ACM: New York, NY, USA, 2009; 189–200, doi:10.1145/1572272.1572295.
39. Briand LC. Novel applications of machine learning in software testing. *Proceedings of the 2008 The Eighth International Conference on Quality Software*, IEEE Computer Society: Washington, DC, USA, 2008; 3–10, doi: 10.1109/QSIC.2008.29.
40. Bowring JF, Rehg JM, Harrold MJ. Active learning for automatic classification of software behavior. *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, ACM: New York, NY, USA, 2004; 195–205, doi:10.1145/1007512.1007539.
41. Briand LC, Labiche Y, Bawar Z. Using machine learning to refine black-box test specifications and test suites. *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, IEEE Computer Society: Washington, DC, USA, 2008; 135–144, doi:10.1109/QSIC.2008.5.
42. Briand LC, Labiche Y, Liu X. Using machine learning to support debugging with tarantula. *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, 2007; 137 –146, doi:10.1109/ISSRE.2007.31.
43. Frounchi K, Briand LC, Grady L, Labiche Y, Subramanyan R. Automating image segmentation verification and validation by learning test oracles. *Inf. Softw. Technol.* Dec 2011; **53**(12):1337–1348, doi:10.1016/j.infsof.2011.06.009.
44. Lo JH. Predicting software reliability with support vector machines. *Computer Research and Development, 2010 Second International Conference on*, 2010; 765 –769, doi:10.1109/ICCRD.2010.144.
45. Wang F, Yao LW, Wu JH. Intelligent test oracle construction for reactive systems without explicit specifications. *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011; 89 –96, doi:10.1109/DASC.2011.39.
46. Madjarov G, Kocev D, Gjorgjevikj D, Deroski S. An extensive experimental comparison of methods for multi-label learning. *Pattern Recognition* 2012; **45**(9):3084 – 3104, doi:http://dx.doi.org/10.1016/j.patcog.2012.03.004. URL http://www.sciencedirect.com/science/article/pii/S0031320312001203, ¡ce:title¿Best Papers of Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA'2011)¡/ce:title¿.
47. Borgwardt KM, Ong CS, Schnauer S, Vishwanathan SVN, Smola AJ, Kriegel HP. Protein function prediction via graph kernels. *Bioinformatics* 2005; **21**(suppl 1):i47–i56, doi:10.1093/bioinformatics/bti1007.