Metamorphic Runtime Checking of Non-Testable Programs

Christian Murphy, Gail Kaiser
Dept. of Computer Science
Columbia University
New York NY 10027
{cmurphy, kaiser}@cs.columbia.edu

ABSTRACT

Challenges arise in assuring the quality of applications that do not have test oracles, *i.e.*, for which it is difficult or impossible to know what the correct output should be for arbitrary input. Recently, metamorphic testing [7] has been shown to be a simple yet effective technique in addressing the quality assurance of these so-called "non-testable programs" [51]. In metamorphic testing, existing test case input is modified to produce new test cases in such a manner that, when given the new input, the function should produce an output that can easily be computed based on the original output. That is, if input x produces output $f(x)$, then we create input x' such that we can predict $f(x')$ based on $f(x)$; if the application does not produce the expected output, then a defect must exist, and either $f(x)$ or $f(x')$ (or both) is wrong.

Previously we have presented an approach called "Automated Metamorphic System Testing" [37], in which metamorphic testing is conducted automatically as the program executes. In the approach, metamorphic properties of the entire application are specified, and then checked after execution is complete. Here, we improve upon that work by presenting a technique in which the metamorphic properties of *individual functions* are used, allowing for the specification of more complex properties and enabling finer-grained runtime checking. Our goal is to demonstrate that such an approach will be more effective than one based on specifying metamorphic properties at the system level, and is also feasible for use in the deployment environment.

This technique, called *Metamorphic Runtime Checking*, is a system testing approach in which the metamorphic properties of individual functions are automatically checked during the program's execution. The tester is able to easily specify the functions' properties so that metamorphic testing can be conducted in a running application, allowing the tests to execute using real input data and in the context of real system states, without affecting those states. We also describe an implementation framework called *Columbus*, and present the results of empirical studies that demonstrate that checking the metamorphic properties of individual functions increases the effectiveness of the approach in detecting defects, with minimal performance impact.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Software Testing, Oracle Problem, Metamorphic Testing

1. INTRODUCTION

It has long been known that there are software applications for which it is difficult to detect subtle errors, faults, defects or anomalies because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. Applications in the fields of scientific calculations, optimizations, machine learning, *etc.* are among those that fall into a category of software that Weyuker describes as *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known"* [51].

One approach to testing such "non-testable programs" [51] is to use a "pseudo-oracle" [12], in which multiple implementations of an algorithm process an input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [26].

In the absence of multiple implementations, metamorphic testing [7] can be used to produce a similar effect. Metamorphic testing is designed as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure, in order to try to find uncovered flaws. Instead of being an approach for test case selection, it is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. In metamorphic testing, if input x produces an output $f(x)$, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation function t, which can then be applied to the input to produce $t(x)$; this transformation then allows us to predict the output $f(t(x))$, based on the (already known) value of $f(x)$. If the output is not as expected (the expectation may be based on some type of equality or semantic equivalence), then a defect must exist. Of course, this can only show the existence of defects and cannot demonstrate their absence, since the correct output cannot be known in advance (and even if the outputs are as expected, both could be incorrect), but metamorphic testing provides a powerful technique to reveal defects in such non-testable programs by use of a built-in pseudo-oracle.

A "metamorphic property" can be defined as the relationship by which the change to the output of a function can be predicted based on a transformation of the input [7]. Consider a function that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result. For instance, one of these metamorphic properties is that permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the devation from the mean would still be the same. Furthermore, other transformations will alter the output, but in a predictable way. For instance, if each value in the set were multipled by 2, then the standard deviation should be twice as much as that of the original set.

Metamorphic properties can exist for an entire application, as well. Consider an application that reads a text file of test scores for students in a class, computes their averages, and uses the function described above to calculate the standard deviation of the averages and determine the students' final grades based on a curve. The application itself has some metamorphic properties, too: permuting the order of the students in the input file should not affect the final grades; nor should multiplying all the scores by 10 (since the students are graded on a curve). These system-level properties are not necessarily the same as those of the constituent functions, but the function-level properties would still be expected to hold.

Previously we have presented an approach called "Automated Metamorphic System Testing" [37], in which metamorphic testing of programs without test oracles is conducted by specifying the metamorphic properties of the entire application. Testing is done automatically as the program executes: the properties are specified prior to execution and then checked after the program is complete. Here, we improve upon that work by presenting a technique in which the metamorphic properties of *individual functions* are used to conduct system testing of software without test oracles, enabling finer-grained runtime checking and increasing the number of test cases; our goal is to demonstrate that such a technique is more effective at identifying defects than simply specifying properties of the application as a whole.

This paper makes three contributions:

1. We introduce a new type of testing called *Metamorphic Runtime Checking*. This is a technique for system testing applications without test oracles in which, rather than specify the metamorphic properties of the application as a whole, we do so for individual functions. While the program is running, we apply functions' "metamorphic properties" to derive new test input for those functions, so that we should be able to predict the corresponding test output; if it is not as predicted then there is a defect in the implementation.

2. We also present an implementation framework called *Columbus* that supports the execution of Metamorphic Runtime Checking from within the context of an application as it runs in the deployment environment, so that real-world inputs can be used to drive the parameters used in metamorphic testing of the individual functions. Columbus conducts the tests with minimal performance overhead, and will ensure that the execution of the tests does not affect the state of the original application process from the users' perspective.

3. Finally, we describe the results of empirical studies of real-world non-testable programs (from the domain of machine learning) to demonstrate the effectiveness of our technique, and compare these results to those in previous work [37] to show that conducting metamorphic testing based on the properties of individual functions shows a 30% improvement in detecting defects over testing based on system-level properties.

2. BACKGROUND

Our work to date has primarily focused on the quality assurance of machine learning applications. As these types of applications become more and more prevalent in various aspects of everyday life [33], it is clear that the dependability of machine learning software takes on increasing importance. The majority of the research effort in the domain of machine learning focuses on building more accurate models that can better achieve the goal of automated learning from the real world. However, to date very little work has been done on assuring the correctness of the software applications that perform machine learning. Formal proofs of an algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is necessary.

Previously we have applied metamorphic testing as part of an approach to testing machine learning applications [34], and identified six categories of metamorphic properties that such applications typically display, which roughly speaking are based on: adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing part of the data; and, adding additional data [36].

Here, we focus on improving the metamorphic testing technique, and demonstrate that system testing that is done by checking the metamorphic properties of individual functions is more effective at detecting defects than checking the properties of entire applications, as we investigated in [37].

2.1 Metamorphic Testing Example

As a more complex example of how metamorphic testing can be used for applications in the domain of machine learning, anomaly-based network intrusion detection systems build up a model of "normal" behavior based on what has previously been observed; this model may be created, for instance, according to the byte distribution of incoming network payloads (since the byte distribution in worms, viruses, *etc.* may deviate from that of normal network traffic [50]). When a new payload arrives, its byte distribution is then compared to that model, and anything deemed anomalous causes an alert. For a particular input, it may not be possible to know *a priori* whether it should raise an alert, since that is entirely dependent on the model. However, if while the program is running we take the new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect ("bug") must exist in the implementation.

Clearly metamorphic testing can be very useful in the absence of an oracle: regardless of the values, if the different outputs for the different inputs are not as expected, then a defect must exist in the implementation. Although the use of these simple relationships for testing numerical functions is not unique to metamorphic testing (*e.g.*, testing based on algebraic properties [11] or programs that can check their work [5]), the approach can be used on a broader domain of any functions that display metamorphic properties, particularly in applications without test oracles.

2.2 Limitations of Previous Approaches

Previous efforts into the automation of metamorphic testing have typically focused on considering the properties of entire applications. Although this has been shown to be simple and effective (since it requires no knowledge of, or even access to, the source code), we intend to show that the technique can be improved using metamorphic testing of the individual functions that display such properties. Intuitively, this makes sense: any property that can be specified for the application itself can at least be specified for the program's entry point function(s), plus the properties of some other functions may be checked, too, so that the number of properties is bound to increase.

Additionally, in order to generate the different test cases, metamorphic testing requires the initial input x and output $f(x)$ values; the inputs could be generated using techniques like equivalence partitioning or random testing [14]. However, inputs chosen using these techniques might miss some defects, since they might not happen to consider a sufficient variety of potential system states or execution paths in the program. Some defects in such systems may only be found under certain application states that may not have been tested prior to deployment: for large, complex software systems, especially if there is no test oracle, it is typically impossible in terms of time and cost to reliably test all possible system states before releasing the product into the field. We require a "perpetual testing" [43] approach that specifically considers the field states that arise in practice, by using real inputs and outputs from actual executions rather than just those generated in the testing lab.

Our goal is to present a technique for performing system testing of applications without test oracles. We demonstrate that such an approach based on using real-world input data to perform runtime checking of the metamorphic properties of individual functions is more effective at detecting defects than checking properties of the program as a whole.

3. APPROACH

To address the limitations described above, we introduce a new technique called *Metamorphic Runtime Checking.* This technique, used for system testing of applications without test oracles, is based on checking the metamorphic properties of individual functions, rather than those of the entire system. This technique can be used to continually test the application as it runs in the deployment environment, to ensure that the properties hold as the program executes.

This must be done in such a manner that the user only

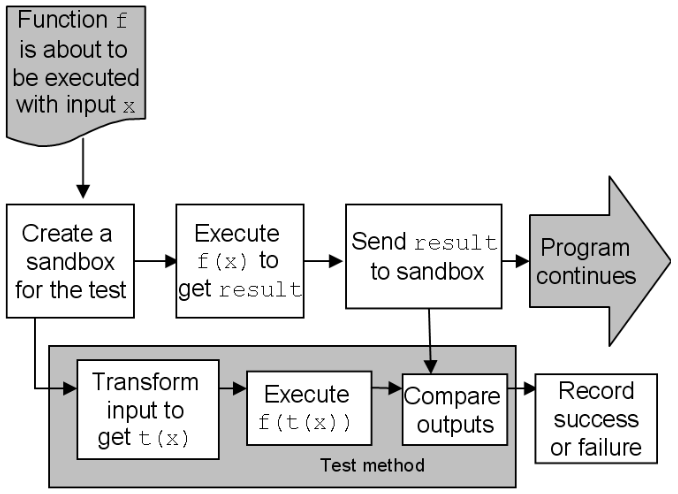
Function f is about to be executed with input x
Create a sandbox for the test
Execute f(x) to get result
Send result to sandbox
Program continues
Transform input to get t(x)
Execute f(t(x))
Compare outputs
Record success or failure
Test method

Figure 1: Model of Metamorphic Runtime Checking

sees the results of the main (original) execution, and not from any function calls that are only for testing purposes. In other words, the user must not observe any modification to the application state; however, the tests themselves **do** need to be able to modify the state because the functions are necessarily being called multiple times, which could have side effects. Thus, the modifications to the state that are caused by the tests must be hidden from the user.

One approach is to run the tests in the same process as the user state and then transactionally roll them back (an idea explored in [29]). Another approach is to create a "sandbox" so that the test function and the original function run in parallel, but in two separate processes that do not affect each other; the sandbox must also make sure that the test function does not affect external entities such as the file system. Metamorphic Runtime Checking uses the sandbox approach, as further described below in Section 3.2.4.

3.1 Model

Metamorphic Runtime Checking is a technique by which tests are to be executed in the running application, using the arguments to instrumented functions as they are called. The arguments are modified according to the specification of the function's metamorphic properties, and the output of the function with the original input is compared to that of the function with the modified input; if the results are not as expected, then a defect has been exposed.

For instance, in the standard deviation example presented above, whenever the function is called, its argument can be passed along to a test method, which will multiply each element in the array by -1 and check that the two calculated output values are equal. This does not require a test oracle for the particular input; the metamorphic relationship specifies its own test oracle. It is true that if the new output is as expected (according to some expectation of equality or sematic equivalence), the results are not necessarily correct, but if the result is **not** as expected, then a defect must exist. This model will allow us to execute tests in the field, within the context of the running application, in applications without a test oracle, by using the metamorphic tests themselves as built-in pseudo-oracles.

In our model of the testing framework, metamorphic tests are logically attached to the functions that they are designed to test. Upon a function's execution, the framework invokes its corresponding test(s) with some probability. In order not to have the user see the effects of the test, the testing framework will execute the metamorphic test in an isolated "sandbox", so that any changes to the state are not reflected in the original process. Additionally, the tests execute in parallel with the application: the test code does not preempt the execution of the application code, which can continue as normal. Figure 1 demonstrates the model we will use for conducting these tests.

3.2 Implementation

In order to facilitate the execution of metamorphic testing in the deployment environment, we require a system that conducts the tests during actual runs of the application, using the same internal state as that of the original function. A system like Skoll [31] is a candidate for something on which to build, but it is primarily intended for execution of regression tests and determining whether builds and installs were successful, and not for testing the system as it runs; other assertion checking techniques (as surveyed in [10]) or monitoring tools (such as Gamma [41]) could be used, but they generally do not allow for calling a function again with different arguments (which we require), and do not safeguard against visible side effects.

For reasons of familiarity and simplicity, the Metamorphic Runtime Checking framework, called *Columbus*, is built upon a testing framework we already had access to that implements what is known as "In Vivo Testing" [35]. Though not specifically focused on metamorphic testing or testing applications without oracles, In Vivo Testing is an approach in which unit tests are executed in the context of the running application without affecting the application state. In the approach, the tests are designed to ensure that properties of given subsystems or units hold true no matter what the application's state is.

3.2.1 In Vivo Testing Overview

In Vivo Testing conducts tests "from within" the running application, using the current accumulated state of the component under test, as opposed to testing from a clean or constructed state, as is typical in unit testing [25]. Developers create tests that are designed ensure that properties of given subsystems or units hold true no matter what the application's state is. In the simplest case, they can be thought of as program invariants and assertions [10], though they go beyond checking the values of individual variables or how variables relate to each other, and focus more on the conditions that must hold after sequences of variable modifications and method calls, without worrying about side effects visible to the user.

An example can be found in Mozilla Firefox. One of the known defects is that attempting to close all other tabs from the shortcut menu of the current tab may fail on Mac OS X when there are more than 20 tabs open.[1] In this case, an In Vivo test designed to run in the field would be one that calls the function to close all other tabs, then checks that no other tabs are open; this sequence should always succeed, regardless of how many tabs were open or what operating system is in use. Particular combinations of execution environment and state may not always be tested in development prior to release of the software, and one way to fully explore whether this property holds in all cases is to test it in the field, as the application is running. Thus, an In Vivo test would be useful in this case.

By combining metamorphic testing and In Vivo Testing, we avoid the need for a test oracle but also gain the benefits of testing in the field: the tests are conducted within the runtime environment, within the context of the application's state. The use of such an approach in the development environment may not reveal defects if the initial test inputs are not sufficient, particularly if the defects only appear in application states that were not or could not have been tested prior to deployment. When we use this approach in the field, we will get a wide range of input values that represent actual usage, as opposed to a smaller set of test cases that are conjured up by developers in the lab.

We have currently implemented the Columbus framework in both Java and C.

3.2.2 Creating Tests

The Columbus framework must be provided with test code that specifies the metamorphic properties to be checked within the running program. This test code would be written by the software developer (as opposed to a third-party developer or the end-user). We currently assume access to the source code, since the instrumentation of the functions is done at compile-time. Given that it is the software developers who will write the tests and instrument the code, we feel that this assumption is reasonable. However, as it may not always be possible or desirable to recompile the code, an approach to dynamically instrumenting the compiled code, such as in Kheiron [18], could be used instead.

To aid in the generation of these tests, as explored in [38], we have created a pre-processor to allow developers to specify metamorphic properties of a function using a special syntax in the comments. Figure 2 shows such properties for an implementation of the sine function, which exhibits two metamorphic properties: $\sin(\alpha) = \sin(\alpha + 2\pi)$ and $\sin(\alpha) = -\sin(-\alpha)$. The parameter "\result" represents the return value of the original function call, so that outputs can be compared; this notation is typical in specification languages such as Java Modeling Language (JML) [27]. These properties can then be used by the pre-processor in the testing framework to generate the test code shown in Figure 3.

```
/*@
  * @meta sine(angle + 2 * M_PI) == \result
  * @meta sine(-angle) == -1 * \result
 */
double sine(double angle) { ...  }
```

Figure 2: Specifying metamorphic properties

```
int __test_sine(double angle, double result) {
   double s0 = sine(angle + 2 * M_PI);
   double s1 = sine(-angle);
   return (s0 == result && s1 == -1 * result);
}
```

Figure 3: Example of a Metamorphic Runtime Checking test generated by the pre-processor

[1]http://www.mozilla.com/en-US/firefox/2.0.0.16/releasenotes/

The testing approach is not limited only to those functions that take input values and return an output, as in the "sine" example; nor is it limited to simple metamorphic properties that can easily be expressed or specified using annotations in the comments. Consider a function *calculate_sum* that determines the sum of the elements in an array referred to by a pointer *p*, and stores that value in a variable *sum*. The developer can then write a test function that permutes the elements in *p*, multiplies them by a random number, calls *calculate_sum*, and checks that the value of *sum* is as expected. Figure 4 shows how the tester could then specify that the metamorphic property of *calculate_sum* is described in the function *__test_calculate_sum*.

```
int* p;
int sum;

/*@
 * @meta __test_calculate_sum()
 */
void calculate_sum() { ...

int __test_calculate_sum() {
   int temp = sum; // remember the old value
   // ...code to randomly permute p...
   int r = rand();
   // ...code to multiply values in p by r...
   calculate_sum();
   return temp == sum * r;

}
```

Figure 4: Example of a manually created Metamorphic Runtime Checking test

Note that the framework ensures that *sum* will already have been modified by the original function call and will have the corresponding result by the time it is accessed in the first line of *__test_calculate_sum*. Additionally, the test is executed in a sandboxed process, so the tester does not have to worry about the fact that *sum* will be overwritten by the test. There are, however, some limitations to the extent of what is sandboxed; see [35] and Section 6 below for details.

3.2.3 *Instrumentation*

Before compiling the source code, the software vendor uses the Columbus pre-processor to first generate test code from the specifications, and then to instrument each annotated function with its corresponding test. During instrumentation, functions to be tested are renamed and wrapped by another function. Figure 5 shows pseudocode for the wrapper of a function f; further details are discussed below.

3.2.4 *Configuration*

At deployment time, the administrator can configure the maximum number of concurrent tests that the system is allowed to execute at any given time. This prevents the testing framework from launching so many simultaneous tests that they flood the CPU and essentially block the main application. The administrator can also set a maximum allowable performance overhead, so that tests will be run only if the overhead of doing so does not exceed the threshold. The system tracks how much time it has spent running tests compared to how much time it has been running application code, and only allows for the execution of tests when the overhead is below the threshold. Alternatively, the administrator can configure the framework so that, for each instrumented function with a corresponding test, there is a probability ρ with which that function's test will be run. The configuration is read at run-time so it can be modified by a system administrator at the customer site if necessary.

```
/* original function */
int __f(int x) { ... }

/* test function */
boolean __test_f(int x, int result) { ... }

/* wrapper function */
int f(int x) {
   int result = __f(x);
   if (should_run_test("f")) {
      create_sandbox_and_fork();
      if (is_test_process()) {
         if (__test_f(x, result) == false) fail();
         else succeed();
         destroy_sandbox();
         exit();
      }
   }
   return result;
}
```

Figure 5: Wrapper of instrumented function

3.2.5 *Execution of Tests*

Once deployed, as shown in Figure 5, the function is first called with its input arguments, and any return value is stored in a variable called "result". The framework then uses the measured performance overhead and/or the probability value ρ for the function to decide whether to execute a test; it also checks whether the maximum allowed number of concurrent tests are currently executing.

When a test is to be executed, a new process is first created as a copy of the original to create a sandbox in which to run the test code, ensuring that any modification to the local process state caused by the test will not affect the "real" application, since the test is being executed in a separate process with separate memory. At this point, the original process continues by returning the result and carrying on as normal; meanwhile, in the test process, the original input and the result of the original function call are passed as arguments to the test function. Within that function, the input can be modified and the outputs can be compared according to the metamorphic properties, without having to worry about changes to the application state. Note that the application and the test run in parallel in two processes: the test does not block normal operation of the application after the sandbox is created. Depending on the configuration and the hardware, the test process may be assigned to a separate CPU or core, so as not to further pre-empt the original process.

In our current implementation of the Columbus framework, we use a process "fork" to create the sandbox, which gives each test process its own memory space to work in, so that it does not alter that of the original process. In our investigations so far, this has been sufficient for our testing purposes, but to ensure that the metamorphic test does not make any changes to the file system, *etc.*, we have begun integration with a thin OS virtualization layer that supports a "pod" (PrOcess Domain) [42] abstraction for creating a virtual execution environment that isolates the process running the test and gives it its own view of the process ID space and a copy-on-write view of the file system. However, the overhead of creating new "pods" may limit the effectiveness of the approach in the general case, so they will likely only be used for tests that actually affect the file system.

When the test is completed, the framework logs whether or not it passed, the process in which the test was run notifies the framework indicating that it is complete (so that the framework can keep track of how many concurrent tests are running), and finally the test process exits.

3.2.6 Handling Test Failure

In the case in which a test fails, the failure is logged to a local file. Additionally, the system administrator can configure what action the system should take when a failure is detected, on a case-by-case basis. In some cases, the administrator may want the system to simply continue to execute normally and ignore the failure; it may be desirable to notify the user of the failed test; and, last, the administrator may choose to have the program terminate.

4. EMPIRICAL STUDIES

To demonstrate that Metamorphic Runtime Checking exhibits an improvement over existing approaches in terms of effectiveness of detecting defects, we performed an empirical study based on the one originally presented using the Automated Metamorphic System Testing (AMST) technique [37]. AMST is based on testing metamorphic properties of the entire application; our goal here is to show that conducting metamorphic testing based on function-level properties will be more effective at detecting defects than using system-level properties.

In these tests, we investigated four real-world "non-testable programs" from the domain of machine learning. The first two are classification algorithms: Support Vector Machines (SVM) [49], as implemented in the Weka [52] 3.5.8 toolkit for machine learning in Java; and C4.5 [45] release 8, which uses a decision tree and is written in C. The third is the ranking algorithm MartiRank [19], also written in C, developed by researchers at Columbia University's Center for Computational Learning Systems (CCLS). Last is the anomaly-based intrusion detection system PAYL [50], implemented in Java.

4.1 Machine Learning Background

In these experiments, we investigated four real-world "non-testable programs" from the domain of machine learning (the authors of this paper were not involved in the development of any of these programs). In *supervised* machine learning, data sets consist of a collection of *examples*, each of which has a number of *attribute* values and, in some cases, a *label*. The examples can be thought of as rows in a table, each of which represents one item from which to learn, and the attributes are the columns of the table. The label indicates how the example is categorized. These applications execute in two phases. The first phase (called the *learning phase*) analyzes a set of *training data*; the result of this analysis is a *model* that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the *classification phase*), the model is applied to another, previously-unseen data set (called the *testing data*) where the labels are unknown.

4.1.1 Support Vector Machines

The Support Vector Machines (SVM) algorithm [49] is one of the more common classification algorithms used in real-world applications, ranging from facial recognition to computational biology [48]. In the learning phase, SVM treats each example from the training data as a vector of N dimensions (since it has N attributes), and attempts to segregate examples from different classes with a hyperplane of N-1 dimensions. In the learning phase, the goal is to find the hyperplane with the maximum margin (distance) between the "support vectors", which are the examples that lie closest to the surface of the hyperplane; the resulting hyperplane is the model. In the classification phase, examples in the testing data are classified according to which "side" of the hyperplane they fall on. The Weka implementation of SVM uses the Sequential Minimal Optimization (SMO) technique [44], which breaks the large quadratic programming optimization problem into smaller problems that can be solved analytically and thus avoids a large matrix computation with limited loss of quality in the results.

4.1.2 C4.5

C4.5 [45] is a very popular algorithm for building decision trees, in which branches represent decisions based on attribute values and leaves represent how the example is to be classified. Like other decision tree classifiers, it takes advantage of the fact that each attribute in the training data can be used to make a decision that splits the data into smaller subsets. During the training phase, for each attribute, C4.5 measures how effective it is to split the data on a particular attribute value, and the attribute with the highest "information gain" (a measure of how well similar labels are grouped together) is the one used to make the decision. The algorithm then continues recursively on the smaller sublists. During classification, the tree is applied to each example, which is classified once it reaches a leaf of the tree.

4.1.3 MartiRank

MartiRank [19] is a ranking algorithm that is used as part of a prototype application for predicting electrical device failures: the examples in the data sets have labels of 0 ("negative example") or 1 ("positive example"), indicating whether the device failed during a particular time period. In the learning phase, MartiRank executes a number of "rounds". In each round the set of training data is broken into sub-lists; there are N sub-lists in the Nth round, each containing $1/N$th of the total number of positive examples. For each sub-list, MartiRank sorts that segment by each attribute, ascending and descending, and chooses the attribute that gives the best "quality". The quality is assessed using a variant of the Area Under the Curve (AUC) [22] calculation that is adapted to ranking rather than binary classification. The model, then, describes for each round how to split the data set and on which attribute and direction to sort each segment for that round. In the second phase, MartiRank applies the segmentation and sorting rules from the model to the testing data set to produce the final ranking.

4.1.4 PAYL

Application	Function	Description	Property
SVM	buildClassifier	Creates a model from a set of instances (training data)	Randomly permuting the order of the instances should yield the same model
SVM	buildClassifier	Creates a model from a set of instances (training data)	Negating the values of the instances should yield the same model but with all values negated
SVM	buildClassifier	Creates a model from a set of instances (training data)	Adding a constant to the values of the instances should yield the same model but with all values increased
SVM	SVMOutput	Computes output (distance from hyperplane) for given instance	If all instances in model have values negated, and given instance does as well, output should stay the same
C4.5	FormTree	Creates a decision tree	Permuting the order of the examples in the training data should not affect the tree
C4.5	FormTree	Creates a decision tree	Multiplying each element in the training data by a constant should yield the same tree, but with the values at decision points also increased
C4.5	FormTree	Creates a decision tree	Adding each element in the training data by a constant should yield the same tree, but with the values at decision points also increased
C4.5	Classify	Classifies an example	Multiplying the values in the example should yield the same classification if the values at decision points are also similarly increased
MartiRank	pauc	Computes the "quality" [22] of a ranking	\result = 1 - reverse ranking
MartiRank	sort_examples	Sorts a set of examples based on a given comparison function	Permuting the order of the elements and negating them returns the same result, but with the elements in the reverse order
MartiRank	sort_examples	Sorts a set of examples based on a given comparison function	Multiplying the elements by a constant returns the same result
MartiRank	insert_score	Inserts a value into an array used to hold top N scores	Calling the function a second time with the same value to be inserted should not affect the array of scores
PAYL	computeTCPLenProb	Computes probability of different lengths of TCP packets	Changing the byte values and permuting their order does not change the results
PAYL	testTCPModel	Returns the distance between an instance and the corresponding "normal" instance in the model	Permuting the order of the elements in the model and multiplying all values by a constant c affects the result by a factor of c

Table 1: Metamorphic properties used in testing

PAYL [50] is an anomaly-based intrusion detection system, and is an example of *unsupervised* machine learning. In PAYL, the training data simply consists of a set of TCP/IP network packets (streams of bytes), without any associated labels or classificiation. During its learning phase, it computes the mean and variance of the byte value distribution for each payload length in order to produce a model of what is considered "normal" network traffic. During the second ("detection") phase, each incoming packet is scanned and its byte value distribution is computed. This new payload distribution is then compared against the model (for that payload length); if the distribution of the new payload is above some threshold of difference from the norm, PAYL flags the packet as anomalous and generates an alert. PAYL may also raise an alert in other circumstances, for instance if the payload length had not been seen in the training data.

4.2 Experimental Setup

In these experiments, mutation testing was used to systematically insert defects into the source code; the goal was to determine whether the mutants could be killed (*i.e.*, whether the defects could be detected) using our approach. Mutation testing has been shown to be a suitable technique for measuring the effectiveness of a test data suite or, as in our case, a testing approach [2]. These mutations fell into three categories: (1) comparison operators were switched to their logical opposites, *e.g.* "less than" was switched to "greater than or equal"; (2) mathematical operators were switched to their opposites, *e.g.* addition was switched to subtraction; and (3) off-by-one errors were introduced for loop variables, array indices, and other calculations that required adjustment by one. All functions in the programs were candidates for the insertion of mutations; each variant that we created had exactly one mutation (*i.e.*, we did not create any program variants with more than one mutation).

To determine which variants were suitable for testing, the output of each was compared to the output of the application with no mutants, which was considered the "gold standard". To obtain this initial output, the same data sets were used as in the experiment with system-level metamorphic properties: for SVM and C4.5, the "iris" data set from the UC-Irvine repository [39] (150 examples, five attributes); for MartiRank, a real-world data set from the electrical device failure application, containing 10,000 examples and 119 attributes; for PAYL, network traffic on the department's LAN (2790 examples). If the outputs of the gold standard and the variant were the same, the mutation would be considered unsuitable for testing, since the mutation may not have been on the execution path, or may have been a "weak mutant" that did not affect the overall output. Additionally, if the mutation yielded a fatal error (crash), an infinite loop, or an output that was clearly wrong (for instance, being nonsensical to someone familiar with the application, or simply being blank), that variant was also discarded since our approach would not be needed to detect such defects.

Once we deteremined which variants could be used for our experiment, we investigated the source code, determined the metamorphic properties of various functions (using the guidelines presented in [36]), and annotated the code with the specifications. Each property was verified with the "gold standard" version to make sure that the property was, in fact, expected to hold. Each of the applications used in this experiment had the same number of metamorphic properties specified as in the original experiment using AMST (four for SVM, C4.5, and MartiRank; two for PAYL); the specific properties are listed in Table 1.

For each variant, we used the Columbus framework to instrument the code and the tests were conducted to see whether Metamorphic Runtime Checking would detect any violation. If so, then the mutant was considered to be killed.

4.3 Findings

The goal of the experiment is to demonstrate that Metamorphic Runtime Checking is more effective in revealing the defects in these applications, by measuring what percentage of the mutants can be killed, and comparing the results to those obtained using Automated Metamorphic System Testing (AMST) in [37].

Tables 2, 3, 4, and 5 summarize the results of our testing of SVM, C4.5, MartiRank, and PAYL respectively. In each table, we specify the type of mutation and the number of mutants that were suitable for use in the testing (*i.e.*, that resulted in a different output compared to the gold standard, and that did not produce an obvious error). The third column shows the total number of distinct mutants killed by the tests when using AMST, and the overall percentage; the last column provides the number detected when using Metamorphic Runtime Checking (abbreviated MRC).

Mutation	**Mutants**	**AMST**	**MRC**
Comparison operators	30	17 (57%)	19 (63%)
Math operators	24	18 (75%)	22 (92%)
Off-by-one	31	31 (100%)	31 (100%)
Total	**85**	**66 (77%)**	**72 (85%)**

Table 2: Results of Mutation Testing for SVM

Mutation	**Mutants**	**AMST**	**MRC**
Comparison operators	8	8 (100%)	8 (100%)
Math operators	15	14 (93%)	14 (93%)
Off-by-one	5	5 (100%)	5 (100%)
Total	**28**	**27 (96%)**	**27 (96%)**

Table 3: Results of Mutation Testing for C4.5

Mutation	**Mutants**	**AMST**	**MRC**
Comparison operators	20	18 (90%)	18 (90%)
Math operators	23	15 (65%)	23 (100%)
Off-by-one	26	17 (65%)	20 (77%)
Total	**69**	**50 (72%)**	**61 (88%)**

Table 4: Results of Mutation Testing for MartiRank

4.4 Discussion and Analysis

Overall, Metamorphic Runtime Checking detected 189 of the 222 defects, compared to 145 detected when using the approach based on system-level metamorphic properties; this is an improvement of 30%. Additionally, as expected, none of the defects detected by the system-level properties were missed by the function-level properties.

Mutation	**Mutants**	**AMST**	**MRC**
Comparison operators	11	1 (9%)	8 (73%)
Math operators	7	0 (0%)	5 (71%)
Off-by-one	22	1 (5%)	16 (73%)
Total	**40**	**2 (5%)**	**29 (73%)**

Table 5: Results of Mutation Testing for PAYL

The improvement shown for the testing of PAYL (Table 5) is admittedly low-hanging fruit. For PAYL, the AMST approach had very little success. In particular, only very basic properties could be used: permuting the ordering of the input data (which were network packets), and permuting the ordering of the bytes within those packet payloads. It was not possible to automate metamorphic tests based on modifying the values of the bytes inside the payloads (say, increasing them), not because of a limitation of the testing tool, but because the application itself only allowed for particular valid inputs that reflected what it considered to be "real" network traffic. However, once we could use Metamorphic Runtime Checking to put the metamorphic tests "inside" the application, we were able to circumvent such restrictions and perform tests using properties of the functions that involved changing the byte values. Thus, we were able to create more complex metamorphic tests that revealed substantially more defects (29 instead of 2). Note that we restricted ourselves to just two metamorphic properties to provide a fair comparison with the original results; adding more properties would presumably have allowed us to detect more defects.

On the flip side, there was not much room for improvement for C4.5 (Table 3); we were unable to devise a test case to detect the last defect, though someone more familiar with the algorithm may be able to (we are admittedly software engineers, not machine learning experts!).

The most interesting result was that in both SVM and MartiRank, **none** of the newly discovered defects were in the functions whose metamorphic properties were being checked. The defects actually existed outside those functions, but put the system into a state in which the metamorphic property of the function would be violated. For example, a particular function in MartiRank took an array of numbers and performed a calculation on them, returning a normalized result (*i.e.*, between 0 and 1). One of the metamorphic properties of that calculation is that reversing the order of the values in the array should produce the "opposite" result, *i.e.*, $f(A)$ = 1 - $f(A')$ where A' is the array in which the values of A are in reverse order. However, a defect in a separate function that dealt with how the array was populated caused this property to be violated because the data structure holding the array itself was in an invalid state, even though the code to perform the calculation was correct.

This metamorphic property was very effective at detecting the defects that had not been found using the system-level approach, especially those related to math operator defects in MartiRank. In particular, the system-level metamorphic properties only considered how the results of different calculations compared to each other, but not their actual values. Consider a simple defect in the system such that the function f described above returns a value that is 0.1 more than it should be. At the system level, the properties that were specified could not access the value returned by f, since the results of the individual calculations were not directly reflected in the program output. Rather, the properties at this level were only influenced by relationships such as: if $f(A) > f(B)$, then $f(A') < f(B')$. Even though the function was producing the wrong result (due to an invalid data structure), this system-level property still held. However, when we used Metamorphic Runtime Checking, we could see that there was a violation in the property of *f*, revealing the defect.

This result demonstrates the real power of our testing technique: without much knowledge of the details of the implementation, we were able to detect many of the defects by simply specifying the expected behavior of particular functions, *even though the defects were not in those functions*; rather, those defects created violations of the metamorphic properties because they put the system into an invalid state. Although we have yet to demonstrate this quantitatively, alternative approaches to detecting such invalid states (such as checking data structure integrity [13] or algebraic specifications [40], further described below) may require more intimate familiarity with the source code, such as the details of pointer references or how variables relate to each other, as opposed to simply specifying how a function should behave when its inputs are modified.

5. PERFORMANCE CONSIDERATIONS

Although Metamorphic Runtime Checking is more effective than AMST at detecting defects, this runtime checking of the metamorphic properties comes at a cost, particularly if the tests are run frequently. In AMST, each metamorphic property is checked exactly once (just at the end of the program execution). In Metamorphic Runtime Checking, though, each property can be checked numerous times, depending on the framework configuration.

During our empirical studies, we measured the performance impact of the Columbus framework. As described above, safeguards have been built into the system to allow for control of the overhead and number of simultaneous test processes; during these tests, however, we removed such restrictions so that we could get a better measurement of the effect of Metamorphic Runtime Checking.

To determine the overhead, we instrumented the functions listed in Table 1 and varied the value probability ρ with which a metamorphic test would be executed while the application ran. Tests were conducted on a server with a dual-core 3GHz CPU running Ubuntu 7.10. Figure 6 shows the results of the experiment, with ρ equal to 0% (with the functions instrumented but no tests executed), 25%, 50%, 75%, and 100% (with all function calls resulting in tests).

The linear nature of the resulting graphs indicates that, as one would expect, the overhead increases linearly with the number of tests that are executed. The slope of the lines results from a combination of the number of tests that are run, and the implementation language: the line for SVM is very steep because many tests were run and the overhead is greater for Java applications; the line for C4.5 is less steep because fewer tests were run and there is less overhead for C.

On average, the performance overhead for the Java applications was around 5.5ms per test; for C, it was only 1.5ms per test. This cost is mostly attributed to the time it takes to create the sandbox and fork the test process.

This impact can certainly be substantial from a percentage overhead point of view if many tests are run in a short-lived program, and some ML programs can run for hours or even days, so care must be taken in configuring the framework. However, for the programs we investigated in our study, the overhead was typically a few seconds, which we consider a small price to pay for detecting that the output of the program was incorrect.

6. RELATED WORK

Others have previously applied metamorphic testing to situations in which there is no test oracle, *e.g.* [8]. In some cases, these works have looked at situations in which there cannot be an oracle for a particular application [9], as in the case of "non-testable programs"; in others, the work has considered the case in which the oracle is simply absent or difficult to implement [6]. However, this previous work did not consider the challenges of automating the process, but rather relied on a tester to manually perform the transformations and comparisons.

Beydeda [4] first brought up the notion of combining metamorphic testing and self-testing components so that an application can be tested at runtime, but did not investigate an implementation or produce any results. Gotleib and Botella [17] have described how the process of metamorphic testing can be conducted automatically, but their work focuses more on the automatic creation of input data that would reveal violations of metamorphic properties, and not on automatically checking that those properties hold after execution. Also, they do not describe any mechanism for addressing performance concerns or for ensuring that the additional invocation of the function or the program is not seen by the user (*i.e.*, their approach was targeted at the development environment, whereas we target both the development environment and - primarily - the deployment environment).

Programming languages such as ANNA [30] and Eiffel [32], as well as C and Java, have built-in support for assertions that allow programmers to check for properties at certain control points in the program. In Metamorphic Runtime Checking, the tests can be considered runtime assertions; however, approaches using assertions typically address how variable values relate to each other, but do not describe the relation between sets of inputs and sets of outputs, as we do in metamorphic testing. Additionally, the assertions in those languages are not allowed to have side effects; in our approach, the tests are allowed to have side effects (in fact they almost certainly will, since the function is called again), but these side effects are hidden from the user. Last, complex assertions (such as checking for data structure integrity [13]) typically pre-empt the application by running sequentially with the rest of the program, whereas in Metamorphic Runtime Checking the program is allowed to proceed while the properties are checked in a parallel process.

Metamorphic properties are similar in some ways to algebraic specifications [11], though algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (*e.g.* $pop(push(X)) == X$ in a Stack), but do not describe how a particular function should react when its input is changed. The runtime checking of algebraic specifications has been explored in [40] and [46], though neither work considered the particular issues that arise from testing without oracles. Even in the cases in which algebraic specifications or formal specification languages (such as Alloy [24], Z [1], *etc.*) are

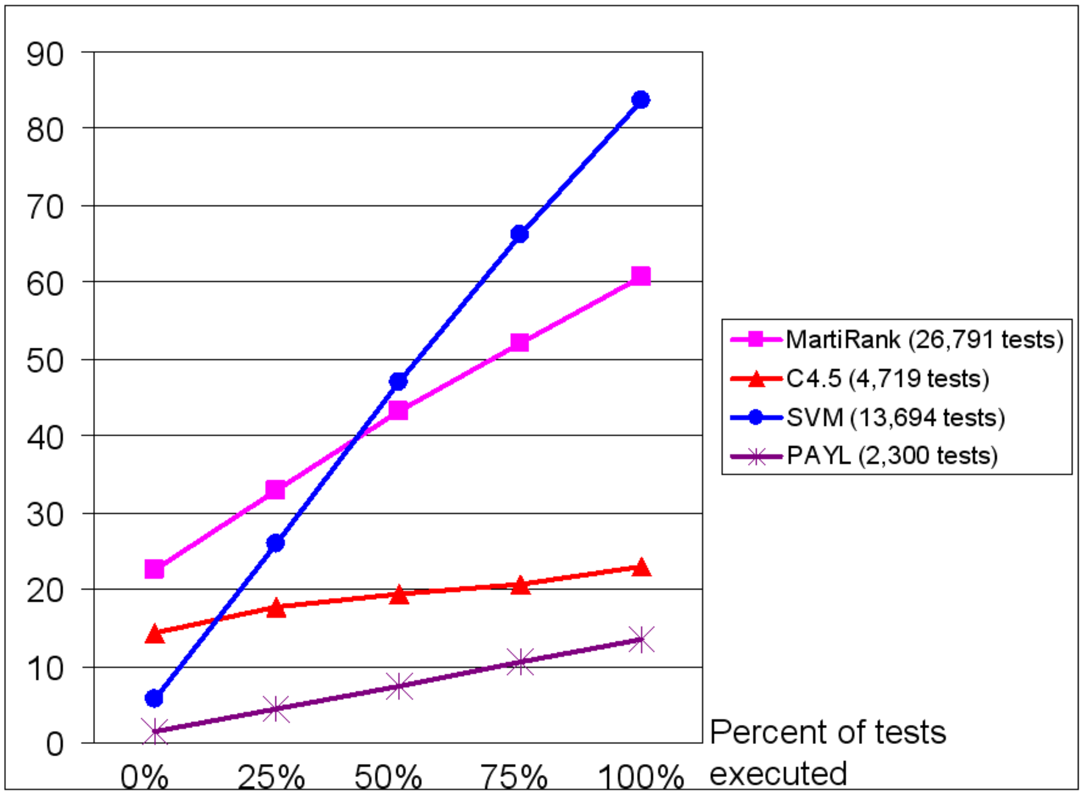

Figure 6: Graph indicating performance overhead caused by different values of ρ for the different applications.

Application	Number of tests	$\rho = 0\%$	$\rho = 25\%$	$\rho = 50\%$	$\rho = 75\%$	$\rho = 100\%$
MartiRank	26,791	22.6s	32.9s	43.2s	52.1s	60.7s
C4.5	4,719	14.3s	17.8s	19.3s	20.6s	22.9s
SVM	13,694	5.7s	26.0s	47.0s	66.1s	83.6s
PAYL	2,300	1.5s	4.4s	7.4s	10.5s	13.5s

Table 6: Results of Performance Tests. The five rightmost columns indicate the time to complete execution with different values of ρ.

used to act as oracles, work to date has focused primarily on consistency checking of abstract data types [47] and has not sought to create (pseudo-)oracles for applications that do not otherwise have them.

The Columbus implementation framework presented here extends our previous work in "In Vivo Testing" [35] in which software tests itself. Other approaches to testing software as it runs in the field include the monitoring, analysis, and profiling of deployed software, as surveyed in [15], and in particular tools like Gamma [41], Skoll [31], and Cooperative Bug Isolation [28]; Columbus differentiates itself from these others by explicitly addressing the problems associated with testing applications without test oracles.

7. LIMITATIONS AND FUTURE WORK

The most critical limitation of the current Columbus implementation is that anything external to the application process itself, *e.g.* database tables, network I/O, *etc.*, is not included in the sandbox and modifications made by a metamorphic test may therefore affect the external state of the original application. As described previously, we are currently looking into integration with a sandboxing solution, and we hope to address these limitations soon.

Another implementation issue is that the test function is called **after** the function to be tested, rather than at the same point in the program execution. This limitation grew out of the necessity to pass the result of the original function call to the test function. Another reason for this implementation decision is that, since the two function calls are in two different processes, challenges would arise in comparing the outputs if the results are pointers, which would point to memory in separate process spaces. The possible side effect of our implementation is that the original function call may alter the system state in such a way that the metamorphic property would not be expected to hold by the time the test function is called, possibly introducing false positives. In our testing, we took precaution to avoid this case, but further investigation needs to be performed to determine how often this problem may arise.

Another limitation of the testing framework is that it uses function calls as the insertion points for metamorphic tests. In our investigation of the MartiRank source code, we noticed that some functions were quite long (over 200 lines) and that we were limited to what sorts of metamorphic properties we could check. Smaller functions may have yielded more opportunities for metamorphic testing. Future work could consider instrumenting the code with tests at arbitrary points, rather than just the start of a function.

Because the approach tests individual functions, it will be clear which function's test failed, so fault localization could start there. However, it may not necessarily be the case that the function itself contains the defect, since the system may be in an invalid state due to a defect in another part of the code (as shown in our experiments). We have begun to investigate other fault localization techniques, though these are currently outside the scope of this particular work.

Additional future work may also include the automatic detection of metamorphic properties, similar to the work that has been done in discovering likely program invariants [16] [21] and algebraic properties [23]. It could be argued that static analysis of the code may be able to determine whether these properties hold, and we have begun preliminary investigations. Further research will be required to determine what are the limits for such approaches when detecting and checking these metamorphic properties.

Finally, the applications we investigated were all deterministic, but approaches like Statistical Metamorphic Testing [20] could easily be incorporated into the framework to address non-determinism. Future work should also explore the application of these techniques to other domains of non-testable programs, such as simulation, optimization, and scientific computing.

8. CONCLUSION

We have introduced *Metamorphic Runtime Checking*, a system testing approach based on the metamorphic properties of individual functions in applications without test oracles. These properties are checked as a program executes in the field, using real inputs from actual program executions. We have also described an implementation framework called *Columbus*, and shown that this approach improves upon other techniques in which metamorphic testing is conducted based on system-level properties.

This work goes beyond applying a system testing approach to individual functions: rather, we use properties of the functions to conduct system testing, and have shown that such properties can detect defects even in functions that are not themselves being tested.

Addressing the testing of applications without oracles has been identified as a future challenge for the software testing community [3]. We hope that our findings here help others who are also concerned with the quality and dependability of such non-testable programs.

9. ACKNOWLEDGMENTS

The authors would like to thank T.Y. Chen, Lori Clarke, Lee Osterweil, Sal Stolfo, Ian Vo, and Junfeng Yang for their guidance and assistance. The authors are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

10. REFERENCES

[1] J. R. Abrial. *Specification Language Z.* Oxford Univ Press, 1980.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments?

In *Proc of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proc. of ICSE Future of Software Engineering (FOSE)*, pages 85–103, 2007.

[4] S. Beydeda. Self-metamorphic-testing components. In *Proc. of the 30th annual computer science and applications conference*, pages 265–272, 2006.

[5] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, Jan. 1995.

[6] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(1):60–80, April-June 2007.

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

[8] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 44(15):923–931, 2002.

[9] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proc. of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 191–195, 2002.

[10] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, May 2006.

[11] W. J. Cody Jr. and W. Waite. *Software Manual for the Elementary Functions*. Prentice Hall, 1980.

[12] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proc. of the ACM '81 Conference*, pages 254–257, 1981.

[13] B. Demsky and M. C. Rinard. Automatic data structure repair for self-healing systems. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[14] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10:438–444, 1984.

[15] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proc. of ISSTA 2004*, pages 65–75, 2004.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely programming invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[17] A. Gotleib and B. Botella. Automated metamorphic testing. In *Proc. of 27th annual international computer software and applications conference (COMPSAC)*, pages 34–40, 2003.

[18] R. Griffith and G. Kaiser. A runtime adaptation framework for native C and bytecode applications. In *3rd IEEE International Conference on Autonomic Computing*, pages 93–103, June 2006.

[19] P. Gross et al. Predicting electricity distribution feeder failures using machine learning susceptibility analysis. In *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, 2006.

[20] 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 Seventh International Conference on Quality Software*, pages 404–409, 2007.

[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[22] J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143:29–36, 1982.

[23] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. of the 17th European Conference on Object-Oriented Programming ECOOP*, 2003.

[24] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[25] JUnit Cookbook. http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[26] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[27] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[28] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the ACM SIGPLAN 2003 conference on programming language design and implementation (PLDI)*, pages 141–154, 2003.

[29] M. Locasto, A. Stavrou, C. F. Cretu, and A. D. Keromytis. From stem to sead: Speculative execution for automated defense. In *Proc of the USENIX Annual Technical Conference*, pages 219–232, 2007.

[30] D. Luckham and F. W. Henke. An overview of ANNA - a specification language for ADA. Technical Report CSL-TR-84-265, Dept. of Computer Science, Stanford Univ., 1984.

[31] A. Memon and A. Porter et al. Skoll: distributed continuous quality assurance. In *Proc. of the 26th ICSE*, pages 459–468, May 2004.

[32] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[33] T. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, 1983.

[34] C. Murphy and G. Kaiser. Improving the dependability of machine learning applications. Technical Report CUCS-49-08, Dept. of Computer Science, Columbia University, 2008.

[35] C. Murphy, G. Kaiser, M. Chu, and I. Vo. Quality assurance of software applications using the in vivo testing approach. In *Proc of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009.

[36] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *Proc. of the 20th international conference on software engineering and knowledge engineering (SEKE)*, pages 867–872, 2008.

[37] C. Murphy, K. Shen, and G. Kaiser. Automated metamorphic system testing. Technical Report CUCS-006-09, Dept. of Computer Science, Columbia University, 2009.

[38] 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 Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009.

[39] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases. University of California, Dept of Information and Computer Science, 1998.

[40] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In *In Proceedings of ICFEM'06, volume 4260 of LNCS*, pages 494–513. Springer-Verlag, 2006.

[41] A. Orso, T. Apiwattanapong, and M.J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European Software Engineering Conf.*, pages 128–137, 2003.

[42] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 361–376, 2002.

[43] L. Osterweil. Perpetually testing software. In *The Ninth International Software Quality Week*, May 1996.

[44] J. Platt. Fast training of support vector machines using sequential minimal optimization. *Advances in Kernel Methods, Support Vector Learning*, 1999.

[45] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[46] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the 1991 international symposium on software testing, analysis, and verification*, pages 123–129, 1991.

[47] S. Sankar, A. Goyal, and P. Sikchi. Software testing using algebraic specification based test oracles. Technical Report CSL-TR-93-566, Dept. of Computer Science, Stanford Univ., 2003.

[48] SVM Application List. http://www.clopinet.com/isabelle/Projects/SVM/.

[49] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[50] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. of the Seventh International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2004.

[51] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[52] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, 2nd Edition.* Morgan Kaufmann, 2005.