Metamorphic Runtime Checking of Non-Testable Programs

CHRISTIAN MURPHY and GAIL KAISER
Columbia University

Challenges arise in assuring the quality of applications that do not have test oracles, *i.e.*, for which it is impossible to know what the correct output should be for arbitrary input. Metamorphic testing has been shown to be a simple yet effective technique in addressing the quality assurance of these "non-testable programs". In metamorphic testing, if test input x produces output $f(x)$, specified "metamorphic properties" are used to create a transformation function t, which can be applied to the input to produce $t(x)$; this transformation then allows the output $f(t(x))$ to be predicted based on the already-known value of $f(x)$. If the output is not as expected, then a defect must exist.

Previously we investigated the effectiveness of testing based on metamorphic properties of the entire application. Here, we improve upon that work by presenting a new technique called *Metamorphic Runtime Checking*, a testing approach that automatically conducts metamorphic testing of individual functions during the program's execution. We also describe an implementation framework called *Columbus*, and discuss 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

Additional Key Words and Phrases: 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 computing, 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"*

The authors are members of the Programming Systems Lab, funded in part by NSF CNS-0905246, CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.
Author's addresses: Columbia University Dept. of Computer Science, 450 CSB, 1214 Amsterdam Avenue, New York NY 10027; email: {cmurphy, kaiser}@cs.columbia.edu.

[Weyuker 1982]. Although some defects in such programs - such as those that cause the program to crash or produce results that are obviously wrong to someone who knows the domain - are easily found, subtle errors in performing calculations or in adhering to specifications can be much more difficult to identify.

One approach to testing such "non-testable programs" [Weyuker 1982] is to use a "pseudo-oracle" [Davis and Weyuker 1981], 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 [Knight and Leveson 1986].

In the absence of multiple implementations, metamorphic testing [Chen et al. 1998] can be used to produce a similar effect. Metamorphic testing is a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any defects, by 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 (or application's) so-called "metamorphic properties" can be used to guide the creation of a transformation function t, which can 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, 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 non-testable programs by use of a built-in pseudo-oracle.

Previously we have investigated the effectiveness of the technique in which metamorphic testing of programs without test oracles is conducted by specifying the metamorphic properties of the entire application [Murphy et al. 2009a]. 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 new technique in which the metamorphic properties of *individual functions* [1] are used to conduct system testing of software that has no test oracle, 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 testing applications without test oracles in which, in addition to specifying the metamorphic properties of the application as a whole, we do so for individual functions as well. While the program is running, we apply functions' metamorphic properties to derive new test input for those functions, so that it should be possible to predict the corresponding test output; if it is not as predicted, then there is a defect in the implementation. This is a new

[1] In this paper, we will use the generic term "functions" to refer to methods, procedures, subroutines, *etc.*, depending on the programming language.

approach that differs from previous work in system-level metamorphic testing in that it is the metamorphic properties of individual functions, not only the entire application, that are specified as well.

(2) We also present a new implementation framework called *Columbus* that supports Metamorphic Runtime Checking by conducting metamorphic tests on the individual functions as the program executes. Columbus conducts the tests with minimal performance overhead, and ensures that the execution of the tests does not affect the state of the original application, so as not to affect further tests in that process.

(3) Finally, we describe the results of new 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 [Murphy et al. 2009a] to show that conducting metamorphic testing based on the properties of individual functions exhibits an improvement in detecting defects over testing based solely on system-level metamorphic properties.

The rest of this paper is organized as follows. Section 2 motivates our work and discusses other approaches to using metamorphic testing to address non-testable programs. Section 3 introduces the Metamorphic Runtime Checking approach, and Section 4 provides some insight into how to devise the properties that can be used in metamorphic testing. Sections 5 and 6 explain the model and architecture of the Columbus framework, respectively. Section 7 discusses the results of our empirical studies, and Section 8 describes the effect the approach has on the time it takes to conduct tests. Section 9 discusses related work, and Section 10 presents limitations and future work. Section 11 concludes.

2. BACKGROUND

2.1 Motivation

This line of research began with work in which we addressed the dependability of a machine learning (ML) application commissioned by a company for potential future experimental use in predicting impending electrical device failures, using historic data of past failures as well as static and dynamic information about the current devices [Murphy and Kaiser 2008]. Classification in the binary sense ("will fail" vs. "will not fail") is not sufficient because, after enough time, every device will eventually fail. Instead, a ranking of the propensity of failure with respect to all other devices is more appropriate. The application uses a variety of ML algorithms in its implementation. We do not discuss the full application further in this paper; see [Gross et al. 2006] for details.

The dependability of the implementation of this system addresses real-world concerns, rather than just academic interest. Although it may be impossible to accurately predict all power outages (which can be due to weather, human error, hungry rats chewing on power cables, *etc.*) there have been cases in which outages might have been prevented via timely maintenance or replacement of devices that were likely to fail, such as the 2008 blackout in Miami [2] and the 2005 blackout in Java

[2]http://www.cnn.com/2008/US/02/26/florida.power/index.html

and Bali.[3] A dependable application in this domain may save money and even lives if it can accurately predict which devices are most likely to fail.

The impact of our research goes far beyond the particular application for which our investigations began. Formal proofs of an ML algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly. As these types of applications become more and more prevalent in society [Mitchell 1983], ensuring their quality becomes more and more crucial. Over fifty different real-world applications, ranging from facial recognition to computational biology, use implementations of the Support Vector Machines [Vapnik 1995] algorithm alone.[4] Additionally, ranking is widely used by Internet search engines (*e.g.*, [Brin and Page 1998]), also apparently using similarly non-testable algorithms. And other ML applications like those used for security and intrusion detection systems are clearly becoming more critical as important data is stored online and attackers seek to access it or gain control of systems.[5] Thus, ensuring the dependability of these sorts of applications takes on increased significance.

2.2 Metamorphic Testing Examples

As described above, metamorphic testing is not a technique that indicates the correctness of individual outputs, but seeks to determine whether a function or application fulfills its expected "metamorphic properties". Even when we cannot know whether the output is correct, we at least know that a violation of the metamorphic properties indicates a defect.

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 [Chen et al. 1998]. 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, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the deviation from the mean would still be the same. Furthermore, other transformations will alter the output, but in a predictable way: if each value in the set were multiplied by 2, then the standard deviation should be twice 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 average, and uses the function described above to calculate the standard deviation of the average and determine the students' final grades based on a curve. Aside from the properties of that function, 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.

As a more complex example, metamorphic testing can be used for applications in the domain of machine learning. For instance, anomaly-based network intru-

[3]http://www.thejakartapost.com/news/2005/08/19/massive-blackout-hits-java-bali.html
[4]http://www.clopinet.com/isabelle/Projects/SVM/applist.html
[5]http://www.sans.org/top20/

sion 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 [Wang and Stolfo 2004]). 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 must exist in the implementation.

Clearly metamorphic testing can be very useful in the absence of an oracle: if the metamorphic property is sound, then regardless of the particular values, if the different outputs for the different inputs do not have the expected relationship, then a defect must exist in the implementation. Although the use of simple relationships for conducting software testing is not unique to metamorphic testing (*e.g.*, testing based on algebraic properties [Cody Jr. and Waite 1980] or programs that can check their work [Blum and Kannan 1995]), the approach can be used on a broader domain of any functions that display metamorphic properties, particularly in applications without test oracles.

2.3 Previous Work in Metamorphic Testing

Others have previously applied metamorphic testing to situations in which there is no test oracle, *e.g.* [Chen et al. 2002a; Zhou et al. 2004]. In some cases, these works have looked at situations in which there cannot be an oracle for a particular application [Chen et al. 2002b], 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 [Chan et al. 2007]. Other efforts into the automation of metamorphic testing have typically focused on either testing individual units in isolation [Chen et al. 2002b], or on system testing by considering the properties of entire applications [Chan et al. 2007; Murphy et al. 2009a; Xie et al. 2009].

Although these works have shown the approach to be simple and effective, we intend to show that the technique can be improved using metamorphic testing of the individual functions that display such properties, within the context of the entire application. That is, as opposed to performing system testing based on properties of the entire application, or by conducting unit testing of isolated pieces of code, we present a technique for testing applications that do not have test oracles by checking the metamorphic properties of its individual functions as the full application runs, instead of testing the functions in isolation. We also present new results demonstrating that system testing that is done in this manner can discover defects that are not found when only using the metamorphic properties of the entire application.

3. APPROACH

Here we introduce a new technique called *Metamorphic Runtime Checking.* This technique, used for system testing of applications that do not have test oracles, is based on checking the metamorphic properties of individual functions, in addition to those of the entire system. This technique is used to ensure that the properties hold as the program executes; any violation of the properties indicates a defect.

In our approach, metamorphic tests are logically attached to the functions that they are designed to test. Upon a function's execution, the corresponding tests are executed as well: 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; at the same time, another test method can multiply each element by 2 and check that the new output is twice as much as the original. This does not require a test oracle for the particular input; the metamorphic relationship specifies its own pseudo-oracle. It is true that if the new output is as expected, the results are not necessarily correct, but if the result is **not** as expected, then a defect must exist.

The steps of the Metamorphic Runtime Checking approach are as follows:

(1) **Identify metamorphic properties.** The tester must first devise the metamorphic properties of the application as a whole, and of the functions that will be used in the testing. We discuss this further in Section 4.

(2) **Specify the properties in the source code.** Once the metamorphic properties have been determined, the source code must be annotated so that the functions' properties can be checked at runtime. In Section 6.2, we describe the mechanism for specifying properties and instrumenting the code in our current implementation of the testing framework.

(3) **Convert the specifications into tests.** In this step, the developer uses a preprocessor to convert the specifications into test methods. These tests will be added to the original source code, as described in Section 6.3.

(4) **Conduct system testing.** Once the metamorphic properties are specified and the code is instrumented with the tests, system testing can commence. The properties are checked as the individual functions execute, and any outputs that deviate from what is expected are indicative of defects in the code.

The approach is not limited only to "pure" functions that do not have side effects, nor to metamorphic properties that depend only on a function's formal parameters as input and its return value as output. In the above example, for instance, if the standard deviation function operates on an array that is part of the system state, and/or has no return value, but rather updates a global variable as a side effect, the metamorphic property can still be checked by considering the array as the "input" and the value of the global variable as the function's "output". Functions need not have any input parameters or any return values to have metamorphic properties: the properties can just specify the expected relationship between the system states before the function call and the states after the call. Details of how this is accomplished in Metamorphic Runtime Checking are described in Section

additive	Increase (or decrease) numerical values by a constant
multiplicative	Multiply numerical values by a constant
permutative	Permute the order of elements in a set
invertive	Reverse the order of elements in a set
inclusive	Add a new element to a set
exclusive	Remove an element from a set

Table I. Classes of metamorphic properties

6.2.

Because Metamorphic Runtime Checking must allow the functions under test to be run multiple times, it must permit these functions to have side effects, but ensure that the side effects of the additional invocations do not affect the process' system state after the test is completed. Clearly, changes to the state caused by calling the function again with modified inputs would be undesirable, and may lead to unexpected system behavior later on. Thus, the metamorphic tests are conducted in a "sandbox", as described below in Section 5.

4. DEVISING METAMORPHIC PROPERTIES

An open issue in the research on metamorphic testing is, "how does one know what the metamorphic properties of the function/application are?" In this section, we describe some guidelines that testers can use when devising the properties used in metamorphic testing.

4.1 Mathematical Properties

Many programs without test oracles rely on mathematical functions (*i.e.*, those that take numerical input and/or produce numerical output), since the point of such programs is to perform calculations, the results of which cannot be known in advance; if they could, the program would not be necessary. In previous work [Murphy et al. 2008], we categorized different classes of metamorphic properties that are common in such mathematical functions, and showed that many applications in the particular domain of interest (machine learning) exhibit these types of properties. The six classes are summarized in Table I; the classes are not meant to imply that the output will not be changed by such transformations, but rather that any change to the output would be predictable.

A simple example (for expository purposes only) of a function that exhibits these different classes of metamorphic properties is one that calculates the sum of a set of numbers. Consider such a function *Sum* that takes as input an array A consisting of n real numbers. Based on the different classes of metamorphic properties listed in Table I, we can derive the following:

(1) **Additive**: If every element in A is increased by a constant c to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A) + nc.

(2) **Multiplicative**: If every element in A is multiplied by a constant c to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A) * c.

(3) **Permutative**: If the elements in A are randomly permuted to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A).

(4) **Invertive**: If the elements in A are placed in reverse order to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A).

(5) **Inclusive**: If a value t is included in the array to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A) + t.

(6) **Exclusive**: If a value t is excluded from the array to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(A) - t.

These are admittedly very trivial examples, but more complex numerical functions that operate on sets or matrices of numbers - such as sorting, calculating standard deviation or other statistics, determining distance in Euclidean space, *etc.* - tend to exhibit similar properties as well. As we pointed out in [Murphy et al. 2009b], such functions are good candidates for metamorphic testing because they are essentially mathematical, and demonstrate well-known properties such as distributivity and transitivity.

4.2 Considerations for General Properties

Although the classes of metamorphic properties listed in Table I have been useful in detecting defects (including the experiment discussed below in Section 7), they are generally only applicable to functions and applications that deal with numerical inputs and outputs. Although non-testable programs tend to fall into this category (machine learning, scientific computing, *etc.*), non-testable programs in domains like computational linguistics and discrete event simulation work with non-numeric data, and these classes of properties may not be applicable.

As a general guideline for creating metamorphic properties, we propose the following. We also provide examples from the domain of discrete event simulation. Such applications can be considered non-testable because the software is written to produce an output (the simulation) that was not already known in advance; if it *were* known in advance, then the simulator would not be necessary.

First, consider the metamorphic properties of **all applications in the given domain**. That is, there may be properties that are shared by all applications that operate in the domain, because of the nature of that domain. For instance, in discrete event simulation, regardless of the particular algorithm, there are generally "resources" that are modeled in the simulation. These resources may be doctors and nurses in a hospital, or assembly line workers in an industrial factory, or postal workers that deliver mail. No matter what algorithm is used, and no matter what is being simulated, all of these share some common metamorphic properties. For instance, increasing the number of resources would be expected to lower each resource's utilization rate, assuming the amount of work to be done remains constant. As another example, if the timing of all events in the simulation is multiplied by a constant factor, then the resource utilization should not change, since the ratio of the time spent working to the total time of the simulation would not be affected (because each are scaled up by the same factor).

Next, consider the properties of **the algorithm chosen to solve a particular problem in that domain**. That is, within the domain, one chooses an algorithm to solve a given problem, and that algorithm will itself have metamorphic properties. For instance, simulators can be used to model the process by which patients are treated in a hospital emergency room [Evans et al. 1996]. The process of simulating a patient's visit to the hospital emergency room might use an algorithm whereby steps and substeps are represented in a tree, and the entire process is essentially a traversal of that tree [Raunak et al. 2009]. This architectural detail leads to numerous metamorphic properties relating to tree traversal: for instance, tree rotation is expected not to change the result of an inorder traversal; also, the tree can be broken into its constituent left and right subtrees, the combined traversal of which should be the same as the traversal of the entire tree. As another example, the selected algorithm may allow for steps of the process to run in parallel; a metamorphic property may be that changing the ordering of the parallel steps in the process specification should not affect the output (since they all execute at the same time, their ordering should not matter).

Then, consider the properties specific to **the implementation of the algorithm** used to solve the problem. A given application that uses the chosen algorithm may have particular metamorphic properties based on features of its implementation, the programming language it uses, how it processes input, how it displays its output, *etc.* For instance, in simulating the operation of a hospital emergency room, the process definition language Little-JIL [Cass et al. 2000] and its corresponding simulator tool (Juliette Simulator, or JSim[6]) may be used to specify the steps that an incoming patient goes through after arriving in the ER. In this implementation, the unique identifiers for the different resources (doctors, nurses, *etc.*) are specified in a plain-text file; since this particular simulator treats all resources as being equal, this implementation exhibits the metamorphic property that permuting the order of the resources in the text file should not affect the simulated process.

Last, consider properties that are applicable only to **the given input** that is being used as part of the test case. Often it is the case that some metamorphic properties of an application will only hold for certain inputs. Consider an input to the hospital emergency room simulation in which the number of resources is sufficiently large so that no patient ever needs to wait. For this particular input, increasing the number of resources should not affect the simulation, since those resources would go unused. But this particular property would not be expected to hold if there were too few resources, of course.

Although the examples provided here are specific to the domain of discrete event simulation (and simulation of a hospital emergency room in particular), it is clear that such an approach could be used in other domains that include non-testable programs, such as machine learning or scientific computing.

In our experience with Metamorphic Runtime Checking, we noted anecdotally that the metamorphic properties of the entire application were often also reflected in one or more individual functions within the code. That is, if one can identify a property of the application, then it is often the case that there will be a function (or perhaps even more than one) that exhibits the same property. Investigation of this phenomenon is outside the scope of this particular work, but during our experiments we observed that often there would be data structures that represented the program input data (either all of it, or a significant part of it); any function that took such a data structure as a parameter was likely to exhibit the same metamorphic property

[6]http://laser.cs.umass.edu/documentation/jsim/

as the entire application since, essentially, the input to the function was the same as the input to the program.

4.3 Automated Detection

We are not aware of any investigation into the automatic discovery of metamorphic properties, though this may be possible by building upon other techniques designed to detect similar characteristics of code. For example, dynamic approaches for discovering likely program invariants (such as Daikon [Ernst et al. 1999] and DIDUCE [Hangal and Lam 2002]) observe program execution and formulate hypotheses of invariants by relaxing different constraints on variables and/or using machine learning techniques to generate rules. Such techniques tend to focus on lower-level implementation details regarding application state and not on higher-level properties regarding function input and output, but could still be used as a basis for a dynamic approach. The automatic detection of metamorphic properties may also build upon the work in the dynamic discovery of algebraic specifications [Henkel and Diwan 2003], though that work has tended to focus on data structures and abstract datatypes, and not on how a function should react when its arguments are modified.

It could be argued that static analysis techniques such as model checking may be able to determine whether these properties hold, though such methods rely on an initial hypothesis of the property to be checked, and are not intended to discover the properties in the first place [Clarke et al. 1999]. Furthermore, many metamorphic properties may be "hidden" within an implementation, and not detectable through analysis of the source code. As a very simple example, a sine function that uses a Taylor series $\sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!)$ obscures the metamorphic property that $\sin(x) = \sin(x + 2\pi)$.

Another approach would be to use machine learning techniques to automatically detect metamorphic properties by considering "similarities" in code. That is, if different pieces of code are known to exhibit a given property, then it may be possible to speculate that "similar" code (by some definition of "similarity") may exhibit the same property. This could also be done using techniques aimed at detecting code clones (*e.g.*, [Gabel et al. 2008]), which typically look for semantic and/or syntactic resemblance, but could conceivably be modified to indicate that two pieces of code exhibit the same metamorphic properties. Such approaches may be feasible in simple cases for the mathematical properties described in Table I, though further investigation is required to determine how the approach fares on arbitrary pieces of more complex code.

4.4 Summary

Though there may be no "silver bullet" when it comes to devising the metamorphic properties of a given function or application, we would argue that in *any* software testing approach, the tester still must have some knowledge or understanding of the program in order to devise test cases. Semantic knowledge of the program or function to be tested is required for writing use cases, devising equivalence classes, creating test input, and designing regression tests [Myers et al. 1979; Everett and McLeod Jr. 2007]. Even purely random testing approaches demand that the tester understand the input and output domain [Hamlet 1994]. Thus, metamorphic test-

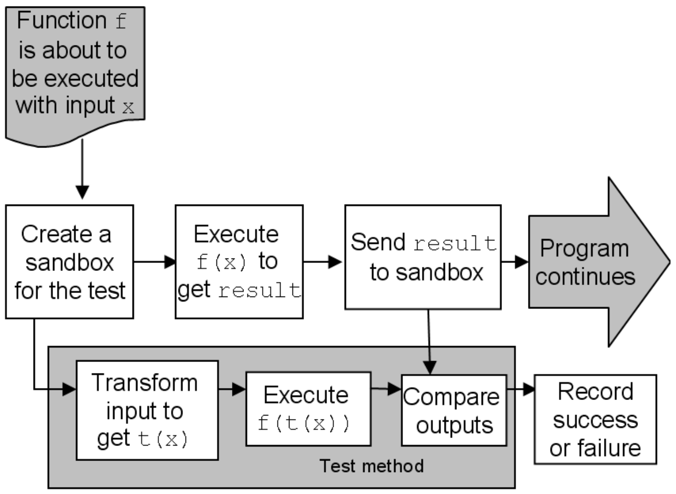
Function f is about to be executed with input x
Create a sandbox for the test
Execute f(x) to get result
Send result to sandbox
Program continues
Transform input to get t(x)
Execute f(t(x))
Compare outputs
Record success or failure
Test method

Fig. 1. Model of Metamorphic Runtime Checking

ing is no different from other black-box techniques in that it is assumed that the tester will have enough knowledge of the code to create test cases (in this case, metamorphic properties), as guided by the program or function specifications and a general understanding of what the code is meant to do.

5. MODEL

Metamorphic Runtime Checking is a technique by which metamorphic tests are 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.

This must be done in such a manner that any changes to the state of the process are the result of only the main (original) function execution, and not from any function calls that are only for testing purposes. In other words, there must not be any observable modification of 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 not affect the application, so that the application can keep executing and testing can continue.

One solution is to run the tests in the same process as the user state and then transactionally roll them back (an idea explored in [Locasto et al. 2007]). Another approach is to create a "sandbox" so that the test function runs in a separate cloned process that does not affect the original; the sandbox must also make sure that the test function does not affect external entities such as the file system.

Additionally, the tests should execute in parallel with the application: the test code should 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.

Note that Metamorphic Runtime Checking does not force the execution of any particular function or corresponding test; rather, it only tests the functions that are actually executed, using the function's arguments and the current system state to check that the metamorphic properties still hold.

6. ARCHITECTURE

In order to facilitate the execution of Metamorphic Runtime Checking, we require a framework that conducts the function-level tests during actual runs of the application, using the same internal state as that of the original function. A system like Skoll [Memon et al. 2004] 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 [Clarke and Rosenblum 2006]) or monitoring tools (such as Gamma [Orso et al. 2003]) 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, we have implemented C and Java prototypes of the Metamorphic Runtime Checking framework, called *Columbus*, based on a framework we already had access to that implements what is known as "In Vivo Testing" [Murphy et al. 2009]. Though not specifically focused on metamorphic testing or testing applications without oracles, In Vivo Testing is an approach in which tests are executed in the context of the running application without affecting the application state. In Vivo tests are designed to ensure that properties of given subsystems or units hold true no matter what the application's state is, particularly as the application runs in the deployment environment ("in vivo") as opposed to the development environment ("in vitro").

6.1 Overview of In Vivo Testing

The foundation of the In Vivo Testing approach is the fact that many (if not all) software products are released into deployment environments with latent defects still residing in them [Pavlopoulou and Young 1999], as well as our claim that these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. For large, complex software systems, it is typically impossible in terms of time and cost to reliably test all possible system states and all possible configuration options before releasing the product into the field. For instance, Microsoft Internet Explorer has over 19 trillion possible combinations of configuration settings [Cohen et al. 2006]. Even given infinite time and resources to test an application and all its configurations, once a product is released, the other software packages on which it depends (libraries, virtual machines, *etc.*) may also be updated; therefore, it would be impossible to test with these dependencies prior to the application's release, because they did not exist yet.

To address this problem, In Vivo Testing is an approach in which software continues to test itself, even in the deployment environment. To accomplish this, tests are conducted "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.[7] Developers create tests that are designed to 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, 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. Additionally, whereas assertion checking is necessarily "read-only", so as not to affect the state of the system, In Vivo tests are allowed to have side effects, but changes to the state of the process are hidden from the end user.

It is important to note that In Vivo tests are not intended to replace unit or integration tests; rather, they are tests designed to run within the context of an executing application, which may be in a previously untested or unanticipated state. Additionally, it is not a *requirement* of In Vivo Testing to run the tests in the field, of course; in Metamorphic Runtime Checking, we primarily intend to run the tests in the development environment, but build upon In Vivo Testing to have the ability to conduct metamorphic testing "from within", *i.e.*, within the context of the application's state, automatically while the application is running.

6.2 Specifying Metamorphic Properties

For the functions that are to be tested, the Columbus framework must be provided with executable test code that specifies the metamorphic properties to be checked within the running program. This test code would be written by the software developer or tester (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 and testers 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 [Griffith and Kaiser 2006], could be used instead.

To aid in the generation of these tests, as presented in [Murphy et al. 2009b], we have created a pre-processor to allow testers to specify metamorphic properties of a function using a special notation 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 [Leavens et al. 2006]. These properties can then be used by the pre-processor in the testing framework to generate the test code shown in Figure 3.

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 tester can then write a test function that permutes the elements in p, multiplies them by a random number, calls *calculate_sum*, and checks that

[7]http://junit.sourceforge.net/doc/cookbook/cookbook.htm

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

Fig. 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);
}
```

Fig. 3. Example of a Metamorphic Runtime Checking test generated by the pre-processor

```
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();
   // check that the property holds
   return temp == sum * r;
}
```

Fig. 4. Example of a manually created Metamorphic Runtime Checking test

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*.

Note that because *__test_calculate_sum* is called after *calculate_sum*, the framework ensures that the variable *sum* will already have been set by the original function call and will have the appropriate result by the time it is accessed in the first line of the test function. 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 additional invocation of *calculate_sum*.

6.3 Instrumentation and Test Execution

Before compiling the source code, the tester 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*.

Once an instrumented function is to be executed, as shown in Figure 5, the function is first called with its input arguments, the "wrapped" original function is called, and any return value is stored in a variable called "result" (line 9). Depending on the configuration, if a test is to be run at this point (line 10), the framework then generates a new process as a copy of the original to create a sandbox in which to run the test code (line 11), ensuring that any modification to the local process state caused by the test will not affect execution of 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 (line 19); meanwhile, in the test process, the original input and the result of the original function call are passed as arguments to the test function (line 13). 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 preempt the original process.

```
 1  /* original function */
 2  int __f(int x) { ... }
 3
 4  /* test function */
 5  boolean __test_f(int x, int result) { ... }
 6
 7  /* wrapper function */
 8  int f(int x) {
 9     int result = __f(x);
10     if (should_run_test("f")) {
11        create_sandbox_and_fork();
12        if (is_test_process()) {
13           if (__test_f(x, result) == false) fail();
14           else succeed();
15           destroy_sandbox();
16           exit();
17        }
18     }
19     return result;
20  }
```

Fig. 5. Wrapper of instrumented function

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. However, to ensure that the metamorphic test does not make any changes to the file system, we have also integrated Columbus with a thin OS virtualization layer that supports a "pod" (PrOcess Domain) [Osman et al. 2002] 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, whereas the overhead of using a "fork" can be as little as a few milliseconds (see Section 8), the overhead of creating new "pods" can be on the order of a few seconds, so they should only be used for tests that actually affect the file system. Testers can indicate that a "pod" is needed for a test via an annotation in the specification of the metamorphic property.

When the test is completed, the framework logs whether or not it passed (Figure 5 lines 13-14), the process in which the test was run notifies the framework indicating that it is complete so that the framework can perform any necessary cleanup (line 15), and finally the test process exits (line 16).

Note that Metamorphic Runtime Checking does not preclude "traditional" metamorphic testing in which the entire application itself is also run a second time with transformed inputs, so that system-level metamorphic properties can also be checked once the process has run to completion. This means that both system-level and function-level properties can be checked during execution, increasing the likelihood of detecting defects.

6.4 Testing in the Deployment Environment

Although Metamorphic Runtime Checking is designed to utilize the metamorphic properties of individual functions to conduct system testing in the lab (*i.e.*, the development environment), the Columbus framework can also be used to conduct tests in the *deployment environment*, as the software runs in the field. In practice, particular combinations of execution environment and state may not always be tested in development prior to release of the software, and one way to further explore whether these properties hold in additional cases is to check them in the field, as the application is running. When testers use this approach in the field, they get a wide range of input values that represent actual usage, as opposed to a smaller set of test cases that are conjured up in the lab.

To support this, the Columbus framework can be configured to limit 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. A system 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 deployment site if necessary.

In the case in which a test fails in the field, 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.

As this paper is focused on detecting defects in applications without test oracles, we do not further explore the implications of testing in the deployment environment, except to discuss the performance overhead in Section 8. See [Murphy et al. 2009] for more details on testing applications in the field using In Vivo Testing.

7. EMPIRICAL STUDIES

In [Murphy and Kaiser 2009], we showed that system-level metamorphic testing is more effective at detecting defects than other approaches, including the use of "pseudo-oracles" or runtime assertion checking; others have independently reported similar findings [Hu et al. 2006]. Here, we improve upon those results, and present a new empirical study that demonstrates that Metamorphic Runtime Checking with function-level properties is more effective than using system-level properties alone.

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). The first two are classification algorithms: Support Vector Machines (SVM) [Vapnik 1995], as implemented in the popular Weka [Witten and Frank 2005] 3.5.8 open-source toolkit for machine learning in Java; and C4.5 [Quinlan 1993] release 8, which uses a decision tree and is written in C. The third is the ranking algorithm MartiRank [Gross et al. 2006], also written in C, developed by researchers at Columbia University's Center for Computational Learning Systems. Last is an application that is not itself an example of ML, but rather uses ML as a critical sub-component: the anomaly-based intrusion detection system PAYL [Wang and Stolfo 2004], implemented in Java by researchers in Columbia University's Intrusion Detection System Lab.

7.1 Machine Learning Background

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 either hidden or absent, with the goal of accurately predicting the label values once they are known. Classification and ranking use supervised machine learning.

Unsupervised ML applications also execute in training and testing phases, but in these cases, the training data sets specifically do not have labels. Rather, an unsupervised ML application seeks to learn properties of the examples on its own, such as the numerical distribution of attribute values or how the attributes relate to each other. This model is then applied to testing data, to determine whether (or to what extent) the same properties exist. Anomaly-detection systems are types of applications that use unsupervised machine learning, as are data mining and collaborative filtering.

7.2 Applications Investigated

C4.5 [Quinlan 1993] is a commonly used 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 [Kullback and Leibler 1951]) is the one used to make the decision. The algorithm then continues recursively on the smaller sublists. During classification, the rules of the tree are applied to each example, which is classified once it reaches a leaf.

MartiRank [Long and Servedio 2005] is a ranking algorithm that is used as part of a prototype application for predicting electrical device failures [Gross et al. 2006]: 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) [Hanley and McNeil 1982] 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. The goal of the second phase is that, once the labels are revealed, positive examples (those with a label of 1) are toward the top of the ranking, and negative examples (with a label of 0) are toward the bottom.

The **Support Vector Machines (SVM)** algorithm [Vapnik 1995] is a commonly-used classification algorithm in real-world applications, ranging from facial recognition to computational biology.[8] 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 that we tested uses the Sequential Minimal Optimization (SMO) technique [Platt 1999], 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.

PAYL [Wang and Stolfo 2004] is an anomaly-based intrusion detection system,

[8]http://www.clopinet.com/isabelle/Projects/SVM/

Application	**Type**	**Metamorphic Property**
C4.5	Permutative	Permuting the order of the examples in the training data should not affect the model
C4.5	Multiplicative	Multiplying each attribute value in the training data by a positive constant (in our case, two) should yield a model in which the values at each decision point have also been multiplied by two
C4.5	Additive	Adding a positive constant (in our case, one) to each attribute value in the training data should yield a model in which the values at each decision point have also been increased by one
C4.5	Invertive	Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same classification
MartiRank	Permutative	Permuting the order of the examples in the training data should not affect the model
MartiRank	Multiplicative	Multiplying each attribute value in the training data by a positive constant (in our case, two) should not affect the model
MartiRank	Additive	Adding a positive constant (in our case, one) to each attribute value in the training data should not affect the model
MartiRank	Invertive	Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same ranking
PAYL	Permutative	Permuting the order of the packets in the training data set should not affect the model
PAYL	Permutative	Permuting the order of the bytes within the payload ("message") in each packet should not affect the model
SVM	Permutative	Permuting the order of the examples in the training data should not affect the model
SVM	Multiplicative	Multiplying each attribute value in the training data by a positive constant (in our case, two) should not affect the model
SVM	Additive	Adding a positive constant (in our case, one) to each attribute value in the training data should not affect the model
SVM	Invertive	Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same classification

Table II. System-level metamorphic properties used in testing

and is an example of an application that uses 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 classification. 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.

ID	App.	Function	Function Description	Metamorphic Property
C1	C4.5	FormTree	Creates a decision tree	Permuting the order of the examples in the training data should not affect the tree
C2	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
C3	C4.5	FormTree	Creates a decision tree	Negating each element in the training data should yield the same tree, but with the values at decision points negated and the comparison operators reversed
C4	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
M1	MartiRank	pauc	Computes the "quality" [Hanley and McNeil 1982] of a ranking	If the ranking is reversed, the quality should be equal to one minus the original result
M2	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
M3	MartiRank	sort_examples	Sorts a set of examples based on a given comparison function	Multiplying the elements by a constant should return the same result
M4	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
P1	PAYL	computeTCP-LenProb	Computes probability of different lengths of TCP packets	Changing the byte values and permuting their order should not change the results
P2	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 should affect the result by a factor of c
S1	SVM	buildClassifier	Creates a model from a set of instances (training data)	Randomly permuting the order of the instances should yield the same model
S2	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
S3	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
S4	SVM	SVMOutput	Computes output (distance from hyperplane) for given instance	If all instances in model have values negated, and given instance does as well, the output should stay the same

Table III. Function-level metamorphic properties used in testing

7.3 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 suitable for evaluation of effectiveness, as experiments comparing mutants to real faults have suggested that mutants are a good proxy for comparisons of testing techniques [Andrews et al. 2005]. 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. Based on our discussions with the researchers who implemented MartiRank and PAYL, we chose these types of mutations because we felt that these represented the types of errors most likely to be made in these types of applications. 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, we used the following data sets: for SVM and C4.5, the "iris" data set from the UC-Irvine repository [Newman et al. 1998] (150 examples, five attributes); for MartiRank, a real-world data set from the prototype electrical device failure application described in Section 2, containing 10,000 examples and 119 attributes; and for PAYL, network traffic on our department's LAN over a one-hour period (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 an "equivalent 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 determined which mutant variants could be used for our experiment, we then used the guidelines set forth in [Murphy et al. 2008] and described above in Section 4 to devise system-level metamorphic properties for the entire applications. These properties are described in Table II. We verified each of these properties with the "gold standard" to ensure that they actually would hold.

Next we investigated the source code, determined the metamorphic properties at the function-level, and verified that they would also hold in the "gold standard". Note that we are not the developers of any of the four applications used in the experiment, so we did not have particularly intimate knowledge of the code (we did, admittedly, have direct access to the developers of MartiRank and PAYL). Even without being very familiar with the code, though, when it came to identifying metamorphic properties for use in the experiment, we were able to use the guidelines described above. The function-level metamorphic properties are listed in Table III; note that these properties are not necessarily the same as the ones for the entire system, they are separate properties that apply to the particular selected functions. Each of the applications used in this experiment had the same number of system-level and function-level metamorphic properties specified (four for SVM, C4.5, and MartiRank; two for PAYL).

For each variant, we first used metamorphic testing with *only* the system-level properties to see whether the violation of the properties would be detected. If so, then the mutant was considered to be killed. We then repeated this for the function-level metamorphic testing using the Columbus framework for Metamorphic Runtime Checking.

Since Metamorphic Runtime Checking extends system-level metamorphic testing by adding the ability to test function-level properties, it is obvious that Metamorphic Runtime Checking will be at least as effective in revealing the defects in these applications. The goal of this experiment is to measure how much more effective it will be, and analyze the results and understand why some defects are more likely to be found than others.

7.4 Findings

Overall, as shown in Table IV, Metamorphic Runtime Checking detected 189 of the 222 defects, compared to 145 detected when using the approach based on system-level metamorphic properties alone; this is an improvement of 30%.

Application	**Total Mutants**	**Mutants killed with System-level properties**	**Mutants killed with Metamorphic Runtime Checking**
C4.5	28	27 (96%)	27 (96%)
MartiRank	69	50 (72%)	61 (88%)
PAYL	40	2 (5%)	29 (73%)
SVM	85	66 (77%)	72 (85%)
Total	**222**	**145 (65%)**	**189 (85%)**

Table IV. Comparison of Results

7.5 Analysis

In this section, we analyze the results of the Metamorphic Runtime Checking experiment and comment on the sensitivity of the results with respect to the different metamorphic properties used.

The improvement shown for the testing of PAYL is admittedly low-hanging fruit, since the system-level 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 conduct system-level metamorphic tests based on modifying the values of the bytes inside the payloads (say, increasing them), not because of a limitation of the approach, but because the application itself only allowed for particular syntactically and semantically 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 27 additional defects.

In SVM, permuting the function input (property S1) was particularly effective in killing the off-by-one mutants not detected by system-level testing. In these mutations, for-loops omitted either the first or last value in an array, thus the mathematical calculations would yield different results because different permutations meant that different elements were being left out. For instance, consider a function $f(A) = \sum_i A_i$, where A is an array of values. One would expect that permuting the order of the elements in A would not affect the result. But clearly if, say, the first element of A is not included in the sum, then permuting the elements will put a different one first, and thus the result will change, in violation of the metamorphic property.

For C4.5 and MartiRank, the metamorphic properties based on multiplication (C2, C4, and M3) were unable to reveal any new defects. The explanation is that for the operations that were changed by the mutations, they would still yield the expected results because of the distributive properties of multiplication. Consider, for an example, a function $f(x,\ y) = x + y$. We would expect it to have the metamorphic property $f(2x,\ 2y) = 2f(x,\ y)$. Now consider a mutation of this function in which the plus sign has been replaced with a minus sign: $f'(x,\ y) = x - y$. Although there is an defect in the code, clearly the metamorphic property $f'(2x,\ 2y) = 2f'(x,\ y)$ still holds; thus, the metamorphic property based on multiplication would not show a violation.

However, this is not necessarily the case for addition (property S3), which does not have similar distributive properties. Consider the same function $f(x,\ y) = x + y$. We would expect it to have the metamorphic property $f(x + 2,\ y + 2) = x + 2 + y + 2 = f(x,\ y) + 4$. Now consider the same mutation of this function in which the plus sign has been replaced with a minus sign: $f'(x,\ y) = x - y$. Now the metamorphic property $f'(x + 2,\ y + 2) = f'(x,\ y) + 4$ no longer holds, because $f'(x + 2,\ y + 2) = x + 2 - (y + 2) = x - y = f'(x,\ y)$; thus, the metamorphic property based on addition would show a violation. This property is more effective in applications based on calculation and computation (like SVM) than it is in applications based on comparison and sorting (like MartiRank).

7.6 Discussion

The most interesting result was that many of the newly discovered defects were in functions other than the ones in which the metamorphic properties were actually being checked (we could not check *all* functions because not all functions have metamorphic properties). 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 instance, the *pauc* function in MartiRank uses an array of numbers (which is part of the application state) and performs a calculation on them to determine the “quality” [Hanley and McNeil 1982] of the ranking, returning a normalized result (*i.e.*, between 0 and 1). One of the metamorphic properties of that calculation (property M1) is that reversing the order of the values in the array should produce the “opposite” result, *i.e.*, $pauc(A) = 1 - pauc(A')$ where A' is the array in which the values of A are in reverse order. However, a defect in a *separate* function that deals 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 in fact correct (we know this, of course, because we know where the defect was seeded in that case).

In particular, the values in the array were being stored in a doubly-linked list, so that MartiRank could calculate the “quality” of the list by looking at it forwards (ascending) and backwards (descending). One mutation in the function that created the linked list caused some of the links to “previous” nodes to point to the wrong ones, as in Figure 6. In this case, traversing the linked list in the forward direction would give *ABCDE*, but backwards would give *EDBCA*. The metamorphic property that $pauc(A) = 1 - pauc(A')$ would only hold if A' were, in fact, the exact opposite

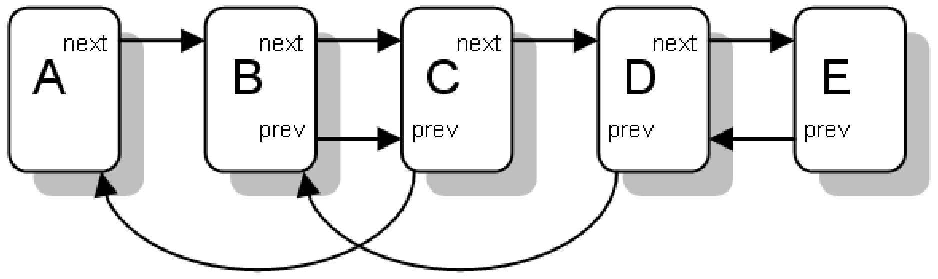

Fig. 6. A doubly-linked list in which elements B, C, and D point to the wrong nodes. The defect that caused this error was detected using a metamorphic property of another function.

ordering of *A*, but clearly in this case it is not. Note that in this case, Metamorphic Runtime Checking detected a defect caused by an invalid application state, and not a defect in the function under test itself (*i.e.*, the one with the metamorphic property).

Property M1 was very effective at detecting the defects that had not been found using the system-level approach (including the one described above), 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 *pauc* function used to calculate the quality of the ranking (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 *pauc*, 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 $pauc(A) > pauc(B)$, then $pauc(A') < pauc(B')$, where *A'* and *B'* are the inverse orderings of *A* and *B*, respectively. Even though the function was producing the wrong result (due to the error in the doubly-linked list, as described above), 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 *pauc*, revealing the defect.

These results demonstrate the real power of our testing technique: without much knowledge of the details of the entire 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 always 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 [Demsky and Rinard 2003] or algebraic specifications [Nunes et al. 2006]) require more intimate familiarity with the source code, such as the details of pointer references or data structures, or dependencies between variables, as opposed to simply specifying how a function should behave when its inputs are modified, using the guidelines described above to identify metamorphic properties.

8. EFFECT ON TESTING TIME

Although Metamorphic Runtime Checking is more effective at detecting defects than metamorphic testing based on system-level properties alone, this checking of the properties comes at a cost, particularly if the tests are run frequently. In system-level metamorphic testing, the program needs to be run once more with the transformed input, and then 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 and the number of times each function is called.

During our empirical studies, we measured the impact of the Columbus framework on the time it took to conduct testing. We instrumented the functions listed in Table III and varied the probability ρ with which a metamorphic test would be executed while the application ran. In order to get a better measurement of the upper bounds of the effect of Metamorphic Runtime Checking, we did not place any limits on the maximum allowable performance overhead or on the number of simultaneous test processes.

Tests were conducted on a server with a dual-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. Table V and Figure 7 show 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 instrumented function calls resulting in tests).

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

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

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 (since Java does not have any "fork" utility, it needed to be implemented via a Java Native Interface call, which added extra overhead); 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

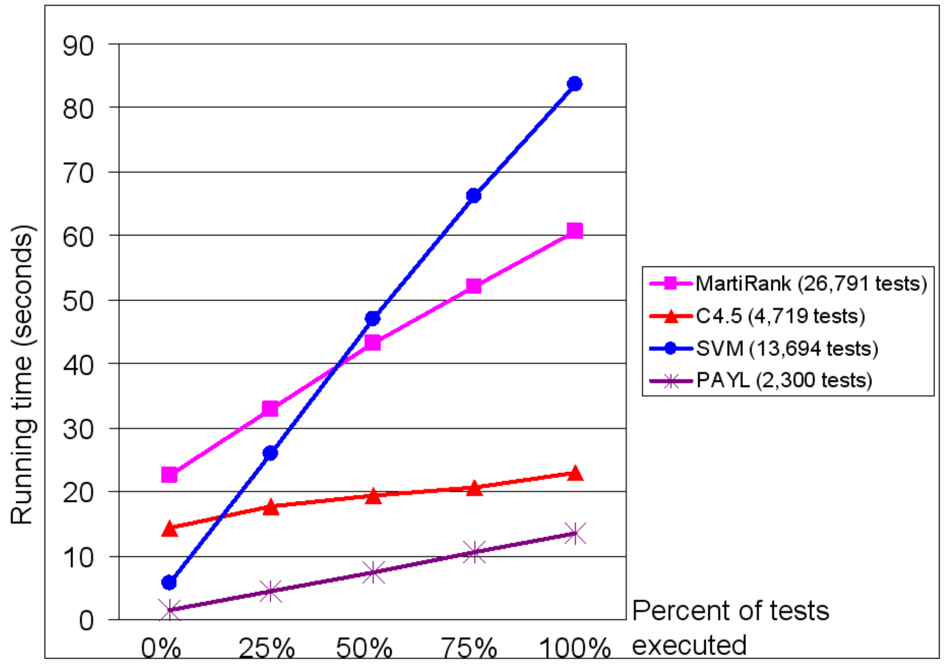

Fig. 7. Graph indicating performance overhead caused by different values of ρ for the different applications.

on the order of a few minutes, which we consider a small price to pay for detecting that the output of the program was incorrect.

9. RELATED WORK

9.1 Testing Non-Testable Programs

Baresi and Young's 2001 survey paper [Baresi and Young 2001] describes various approaches to testing software without a test oracle. Some of those approaches are described here:

Programming languages such as ANNA [Luckham and Henke 1984] and Eiffel [Meyer 1992], 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 [Leveson et al. 1990; Rosenblum 1995]. 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 [Demsky and Rinard 2003]) typically preempt 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 parallel.

Extrinsic Interface Contracts are similar to assertions except that, rather than embedding the specifications in the source code, they are kept separate from the implementation, as in the programming language ADL [Sankar and Hayes 1994]. Another example would be algebraic specifications [Cody Jr. and Waite 1980], which are similar to metamorphic properties, 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 [Nunes et al. 2006] and [Sankar 1991], though neither work considered the particular issues that arise from testing without oracles, or issues related to side effects. Even in the cases in which algebraic specifications are used to act as oracles, work to date has focused primarily on consistency checking of abstract data types [Sankar et al. 2003] and has not sought to create (pseudo-)oracles for applications and functions that do not otherwise have them.

Formal specification languages like Z [Abrial 1980] or Alloy [Jackson 2002] can be used to declare the specific properties of the application, typically in advance of the implementation to communicate intended behavior to the developers. However, Baresi and Young point out that a challenge of using specification languages as oracles is that "*effective procedures for evaluating the predicates or carrying out the computations they describe are not generally a concern in the design of these languages*", *i.e.*, the language may not be powerful enough to describe how to know whether the implementation is meeting the specification. Although previous work has demonstrated that formal specification-based assertions can be effective in acting as test oracles [Coppit and Haddox-Schatz 2005], the specifications need to be complete in order to be of practical use in the general case, as pointed out in [Sankar et al. 2003].

Using debugging or trace tools to observe the execution of an application may indicate whether or not it is functioning correctly, if for instance it is conforming to certain properties (like a sequence of execution calls or a change in variable values) that are believed to be related to correct behavior; or, conversely, to see if it is *not* conforming to these properties. We have, in fact, investigated this technique previously with some success [Murphy and Kaiser 2008], but noted that often the creation of an oracle to tell if the trace is correct can be just as difficult as creating an oracle to tell if the output is correct in the first place, assuming it is even possible at all.

9.2 Testing Machine Learning Applications

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (*e.g.*, [Briand 2008], [Cheatham et al. 1995], [Zhang and Tsai 2003], *etc.*), we are not currently aware of any work in the reverse sense: applying software testing techniques to machine learning applications, particularly those that have no reliable test oracle. Orange [Demsar et al.] and WEKA [Witten and Frank 2005] are two of several frameworks that aid ML developers, but the testing functionality they provide is focused on comparing the quality of the results, and not evaluating the "correctness" of the implementations. Repositories of "reusable" data sets have been collected (*e.g.*, the UCI Machine Learning Repository [Newman et al. 1998]) for the purpose of comparing result quality, *i.e.*, how accurately the algorithms predict, but not for the software engineering sense of testing: an implementation may predict very well, but still have defects.

Testing of intrusion detection systems [Mell et al. 2003; Nicholas et al. 1996], intrusion tolerant systems [Madan et al. 2004], and other security systems [Balzarotti et al. 2008] has typically addressed quantitative measurements like overhead, false alarm rates, or ability to detect zero-day attacks, but does not seek to ensure that the implementation is free of defects, as we do here.

9.3 Metamorphic Testing

Beydeda [Beydeda 2006] 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 consider the effectiveness on testing applications without oracles. Gotleib and Botella [Gotleib and Botella 2003] 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.

Test case selection (*i.e.*, choosing the combination of metamorphic properties and functions to test so that defects are most likely to be revealed) in metamorphic testing is detailed in [Chen et al. 2004] and [Mayer and Guderlei 2006]. This can be based on common metrics like statement or path coverage, or test case dominance (where one test case subsumes others). However, test case selection is not necessarily an issue here, since the Metamorphic Runtime Checking approach conducts tests based on the functions that are actually called while the program is running, and does not seek to drive the execution of particular functions.

9.4 Self-Testing Software

While the notion of "self-checking software" is by no means new [Yau and Cheung 1975], much of the recent work in self-testing components has focused on COTS component-based software. This stems from the fact that users of these components often do not have the components' source code and cannot be certain about their quality. Approaches to solving this problem include using retrospectors [Liu and Richardson 1998] to record testing and execution history and make the information available to a software tester, and "just-in-time testing" [Liu and Richardson 2002] to check component compatibility with client software. Work in "built-in-testing" [Wang et al. 2000] has included investigation of how to make components testable [Beydeda 2005; Beydeda and Gruhn 2003; Brenner et al. 2007; Mariani et al. 2004], and frameworks for executing the tests [Denaro et al. 2003; Mao et al. 2007; Merdes et al. 2006], including those in Java programs [Deveaux et al. 2001], or through the use of aspect-oriented programming [Mao 2007]. However, none of these address the issue of testing applications without test oracles, or of using properties of the individual functions to perform system testing.

Other "perpetual testing" [Osterweil 1996] approaches to testing software as it runs in the field include the monitoring, analysis, and profiling of deployed software, as surveyed in [Elbaum and Hardojo 2004], and in particular tools like Gamma [Orso et al. 2003], Skoll [Memon et al. 2004], and Cooperative Bug Isolation [Liblit et al. 2003]; Columbus differentiates itself from these others by explicitly addressing the problems associated with testing applications without test oracles.

10. 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. Although this appears to limit the usefulness of the approach, we note, however, that in our testing, the current sandbox implementation (which provides the test process with its own memory space and own view of the file system) would have been sufficient for the applications we tested: none of the applications used an external database, and the metamorphic properties that were checked in the network intrusion detection system PAYL did not involve network I/O. For database-driven applications, it may be possible to automate the creation of sandboxed database tables using copy-on-write technology (as in Microsoft SQL Server [9]) or "safe" test case selection techniques that ensure that there will be no permanent changes to the database state as a result of the tests [Willmor and Embury 2005; 2006]; we leave these as future work.

Another implementation issue is that the test functions are 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 functions. Another reason for this implementation decision is that, since the function calls are in 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, none of the selected metamorphic properties fell into this trap, but further investigation needs to be performed to determine how often this problem may arise.

One possible direction for future work lies in fault localization. Because the approach tests individual functions, it will be clear which function's test revealed a violation of its metamorphic properties. 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 based on taking process checkpoints, though these are currently outside the scope of this particular work.

Finally, the applications we investigated were all deterministic, but approaches like Statistical Metamorphic Testing [Guderlei and Mayer 2007] and Heuristic Meta-

[9]http://msdn.microsoft.com/en-us/library/ms175158.aspx

morphic Testing [Murphy et al. 2009a] could 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 discrete event simulation, optimization, and other fields of scientific computing.

11. CONCLUSION

We have introduced *Metamorphic Runtime Checking*, a new system testing approach based on checking the metamorphic properties of individual functions in applications without test oracles. 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 [Bertolino 2007]. We hope that our findings here help others who are also concerned with the quality and dependability of such non-testable programs.

ACKNOWLEDGMENTS

The authors would like to thank T.Y. Chen, Lori Clarke, Lee Osterweil, Sal Stolfo, and Junfeng Yang for their guidance and assistance. Sahar Hasan, Lifeng Hu, Kuang Shen, and Ian Vo all contributed to the implementation of the Columbus framework.

REFERENCES

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

Andrews, J. H., Briand, L. C., and Labiche, Y. 2005. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th International Conference on Software Engineering (ICSE)*. 402–411.

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proc. of the 2008 IEEE Symposium on Security and Privacy*. 387–401.

Baresi, L. and Young, M. 2001. Test oracles. Tech. Rep. CIS-TR01 -02, Dept. of Computer and Information Science, Univ. of Oregon.

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

Beydeda, S. 2005. Research in testing COTS components - built-in testing approaches. In *Proc. of the 3rd ACS/IEEE International Conference on Computer Systems and Applications.*

Beydeda, S. 2006. Self-metamorphic-testing components. In *Proc. of the 30th Annual Computer Science and Applications Conference (COMPSAC)*. 265–272.

Beydeda, S. and Gruhn, V. 2003. The self-testing cots components (STECC) strategy - a new form of improving component testability. In *Proc. of the Seventh IASTED International Conference on Software Engineering and Applications*. 222–227.

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

Brenner, D., Atkinson, C., Paech, B., Malaka, R., Merdes, M., and Suliman, D. 2007. Reducing verification effort in component-based software engineering through built-in testing. *Information System Frontiers 9,* 2-3, 151–162.

Briand, L. 2008. Novel applications of machine learning in software testing. In *Proc. of the Eighth International Conference on Quality Software*. 3–10.

Brin, S. and Page, L. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh International Conference on the World Wide Web*. 107–117.

Cass, A. G., Lerner, B. S., McCall, E. K., Osterweil, L. J., Sutton Jr., S. M., and Wise, A. 2000. Little-JIL/Juliette: A process definition language and interpreter. In *Proc. of the 22nd International Conference on Software Engineering (ICSE)*. 754–757.

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

Cheatham, T. J., Yoo, J. P., and Wahl, N. J. 1995. Software testing: a machine learning experiment. In *Proc. of the ACM 23rd Annual Conference on Computer Science*. 135–141.

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

Chen, T. Y., Huang, D. H., Tse, T. H., and Zhou, Z. Q. 2004. Case studies on the selection of useful relations in metamorphic testing. In *Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. 569–583.

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

Chen, T. Y., Tse, T. H., and Zhou, Z. Q. 2002b. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 191–195.

Clarke, E. M., Grumberg, O., and Peled, D. 1999. *Model Checking.* MIT Press.

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

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

Cohen, M. B., Snyder, J., and Rothermel, G. 2006. Testing across configurations: implications for combinatorial testing. In *Proc. of the Second Workshop on Advances in Model-based Software Testing*. 1–9.

Coppit, D. and Haddox-Schatz, J. 2005. On the use of specification-based assertions as test oracles. In *Proc. of the 29th Annual IEEE/NASA Software Engineering Workshop*.

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

Demsar, J., Zupan, B., and Leban, G. Orange: From experimental machine learning to interactive data mining. [www.ailab.si/orange], Faculty of Computer and Information Science, University of Ljubljana.

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

Denaro, G., Mariani, L., and Pezz'e, M. 2003. Self-test components for highly reconfigurable systems. In *Proc. of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03), vol. ENTCS 82(6)*.

Deveaux, D., Frison, P., and Jezequel, J.-M. 2001. Increase software trustability with self-testable classes in Java. In *Proc. of the 2001 Australian Software Engineering Conference*. 3–11.

Elbaum, S. and Hardojo, M. 2004. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 65–75.

ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. 1999. Dynamically discovering likely programming invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*. 213–224.

EVANS, G. W., GOR, T. B., AND UNGER, E. 1996. A simulation model for evaluating personnel schedules in a hospital emergency department. In *Proc. of the 28th Conference on Winter Simulation*. 1205–1209.

EVERETT, G. D. AND MCLEOD JR., R. 2007. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Press.

GABEL, M., JIANG, L., AND SU, Z. 2008. Scalable detection of semantic clones. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*. 321–330.

GOTLEIB, A. AND BOTELLA, B. 2003. Automated metamorphic testing. In *Proc. of 27th Annual International Computer Software and Applications Conference (COMPSAC)*. 34–40.

GRIFFITH, R. AND KAISER, G. 2006. A runtime adaptation framework for native C and bytecode applications. In *Proc. of the 3rd IEEE International Conference on Autonomic Computing*. 93–103.

GROSS, P., BOULANGER, A., ARIAS, M., WALTZ, D., LONG, P. M., LAWSON, C., ANDERSON, R., KOENIG, M., MASTROCINQUE, M., FAIRECHIO, W., JOHNSON, J. A., LEE, S., DOHERTY, F., AND KRESSNER, A. 2006. Predicting electricity distribution feeder failures using machine learning susceptibility analysis. In *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*.

GUDERLEI, R. AND MAYER, J. 2007. 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*. 404–409.

HAMLET, D. 1994. Random testing. *Encyclopedia of Software Engineering*, 970–978.

HANGAL, S. AND LAM, M. S. 2002. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*. 291–301.

HANLEY, J. A. AND MCNEIL, B. J. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology 143*, 29–36.

HENKEL, J. AND DIWAN, A. 2003. Discovering algebraic specifications from Java classes. In *Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP)*.

HU, P., ZHANG, Z., CHAN, W. K., AND TSE, T. H. 2006. An empirical comparison between direct and indirect test result checking approaches. In *Proc. of the 3rd International Workshop on Software Quality Assurance*. 6–13.

JACKSON, D. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology 11,* 2, 256–290.

KNIGHT, J. AND LEVESON, N. 1986. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering 12,* 1, 96–109.

KULLBACK, S. AND LEIBLER, R. A. 1951. On information and sufficiency. *The Annals of Mathematical Statistics 22,* 1, 79–86.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes 31,* 3 (May), 1–38.

LEVESON, N. G., CHA, S. S., KNIGHT, J. C., AND SHIMEALL, T. J. 1990. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering 16,* 4 (April), 432–443.

LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. 2003. Bug isolation via remote program sampling. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 141–154.

LIU, C. AND RICHARDSON, D. 1998. Software components with retrospectors. In *Proc. of the International Workshop on the Role of Software Architecture in Testing and Analysis*. 63–68.

LIU, C. AND RICHARDSON, D. J. 2002. RAIC: Architecting dependable systems through redundancy and just-in-time testing. In *Proc. of the ICSE Workshop on Architecting Dependable Systems (WADS)*.

Locasto, M., Stavrou, A., Cretu, G. F., and Keromytis, A. D. 2007. From STEM to SEAD: Speculative execution for automated defense. In *Proc. of the USENIX Annual Technical Conference*. 219–232.

Long, P. and Servedio, R. 2005. Martingale boosting. In *Proc. of the 18th Annual Conference on Computational Learning Theory (COLT)*. 79–84.

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

Madan, B., Goševa-Popstojanova, K., Vaidyanathan, K., and Trivedi, K. S. 2004. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Performance Evaluation Journal 56,* 1-4, 167–186.

Mao, C. 2007. AOP-based testability improvement for component-based software. In *Proc. of the 31st Annual International COMPSAC, vol. 2*. 547–552.

Mao, C., Lu, Y., and Zhang, J. 2007. Regression testing for component-based software via built-in test design. In *Proc. of the 2007 ACM Symposium on Applied Computing*. 1416–1421.

Mariani, L., Pezz'e, M., and Willmor, D. 2004. Generation of integration tests for self-testing components. In *Proc. of FORTE 2004 Workshops, Lecture Notes in Computer Science, Vol.3236*. 337–350.

Mayer, J. and Guderlei, R. 2006. An empirical study on the selection of good metamorphic relations. In *Proc. of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*. 475–484.

Mell, P., Hu, V., Lippmann, R., Haines, J., and Zissman, M. 2003. An overview of issues in testing intrusion detection systems. Tech. Rep. Tech. Report NIST IR 7007, National Institute of Standard and Technology.

Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D., and Natarajan, B. 2004. Skoll: distributed continuous quality assurance. In *Proc. of the 26th International Conference on Software Engineering (ICSE)*. 459–468.

Merdes, M., Malaka, R., Suliman, D., Paech, B., Brenner, D., and Atkinson, C. 2006. Ubiquitous RATs: how resource-aware run-time tests can improve ubiquitous software systems. In *Proc. of the 6th International Workshop on Software Engineering and Middleware*. 55–62.

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

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

Murphy, C. and Kaiser, G. 2008. Improving the dependability of machine learning applications. Tech. Rep. CUCS-49-08, Dept. of Computer Science, Columbia University.

Murphy, C. and Kaiser, G. 2009. Empirical evaluation of approaches to testing applications without test oracles. Tech. Rep. CUCS-039-09, Dept. of Computer Science, Columbia Univ.

Murphy, C., Kaiser, G., Chu, M., and Vo, I. 2009. 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)*.

Murphy, C., Kaiser, G., Hu, L., and Wu, L. 2008. Properties of machine learning applications for use in metamorphic testing. In *Proc. of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 867–872.

Murphy, C., Shen, K., and Kaiser, G. 2009a. Automated metamorphic system testing. In *Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA)*.

Murphy, C., Shen, K., and Kaiser, G. 2009b. 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)*.

Myers, G. J., Badgett, T., Thomas, T. M., and Sandler, C. 1979. *The art of software testing.* Wiley.

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

Nicholas, J. P., Zhang, K., Chung, M., Mukherjee, B., and Olsson, R. A. 1996. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering 22,* 10, 719–729.

Nunes, I., Lopes, A., Vasconcelos, V., Abreu, J., and Reis, L. S. 2006. Checking the conformance of Java classes against algebraic specifications. In *Proc. of the International Conference on Formal Engineering Methods (ICFEM), volume 4260 of LNCS*. Springer-Verlag, 494–513.

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

Osman, S., Subhraveti, D., Su, G., and Nieh, J. 2002. 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)*. 361–376.

Osterweil, L. 1996. Perpetually testing software. In *Proc. of the The Ninth International Software Quality Week*.

Pavlopoulou, C. and Young, M. 1999. Residual test coverage monitoring. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*. 277–284.

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

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

Raunak, M., Osterweil, L., Wise, A., Clarke, L., and Henneman, P. 2009. Simulating patient flow through an emergency department using process-driven discrete event simulation. In *Proc. of the 2009 ICSE Workshop on Software Engineering in Health Care*. 73–83.

Rosenblum, D. S. 1995. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering 21,* 1 (January), 19–31.

Sankar, S. 1991. Run-time consistency checking of algebraic specifications. In *Proc. of the 1991 International Symposium on Software Testing, Analysis, and Verification (TAV)*. 123–129.

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

Sankar, S. and Hayes, R. 1994. Adl: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices 29,* 8 (August), 13–21.

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

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

Wang, Y., Patel, D., King, G., Court, I., Staples, G., Ross, M., and Fayad, M. 2000. On built-in test reuse in object-oriented framework design. *ACM Computing Surveys 32,* 1 (March).

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

Willmor, D. and Embury, S. M. 2005. A safe regression test selection technique for database-driven applications. In *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM)*. 421–430.

Willmor, D. and Embury, S. M. 2006. An intensional approach to the specification of test cases for database applications. In *Proc. of the 28th International Conference on Software Engineering (ICSE)*. 102–111.

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

Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., and Chen, T. Y. 2009. Application of metamorphic testing to supervised classifiers. In *Proc. of the 9th International Conference on Quality Software (QSIC)*.

Yau, S. S. and Cheung, R. C. 1975. Design of self-checking software. In *Proc. of the International Conference on Reliable Software*. 450–455.

Zhang, D. and Tsai, J. J. P. 2003. Machine learning and software engineering. *Software Quality Control 11,* 2 (June), 87–119.

Zhou, Z. Q., Huang, D. H., Tse, T. H., Yang, Z., Huang, H., and Chen, T. Y. 2004. Metamorphic testing and its applications. In *Proc. of the 8th International Symposium on Future Software Technology (ISFST 2004)*.

Received Oct 2009;