SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–35
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Automatic Detection of Defects in Applications without Test Oracles

Christian Murphy* and Gail Kaiser

Department of Computer Science, Columbia University, New York NY 10027

SUMMARY

In application domains that do not have a test oracle, such as machine learning and scientific computing, quality assurance is a challenge because it is difficult or impossible to know in advance what the correct output should be for general input. Previously, metamorphic testing has been shown to be a simple yet effective technique in detecting defects, even without an oracle. In metamorphic testing, the application's "metamorphic properties" are used to modify existing test case input to produce new test cases in such a manner that, when given the new input, the new output can easily be computed based on the original output. If the new output is not as expected, then a defect must exist. In practice, however, metamorphic testing can be a manually intensive technique for all but the simplest cases. The transformation of input data can be laborious for large data sets, and errors can occur in comparing the outputs when they are very complex. In this paper, we present a tool called Amsterdam that automates metamorphic testing by allowing the tester to easily set up and conduct metamorphic tests with little manual intervention, merely by specifying the properties to check, configuring the framework, and running the software. Additionally, we describe an approach called Heuristic Metamorphic Testing, which addresses issues related to false positives and non-determinism, and we present the results of new empirical studies that demonstrate the effectiveness of metamorphic testing techniques at detecting defects in real-world programs without test oracles.
Copyright © 2010 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: software testing; oracle problem; metamorphic testing; heuristic testing; non-testable programs

1. INTRODUCTION

Despite advances in formal methods and static analysis techniques, software testing still remains the primary mechanism by which software defects are found. This involves identifying the pieces of code to test, deriving test inputs, executing the code with those inputs, and then using a "test oracle" [1] to determine whether (or to what extent) the output matches what was expected. The tester assumes that this test oracle is reliable, in that it is both sound and complete. While certain defects, such as those leading to crashes or infinite loops, are easily observed, the creation of reliable test oracles to indicate that the program output is correct is a problem that researchers have investigated for many years [2].

Additional challenges arise in testing applications in the fields of scientific computing, simulation, optimization, machine learning, etc. because often there is no reliable test oracle to indicate what the correct output should be for arbitrary input. In some cases, it may be impossible to know the program's correct output *a priori*; in other cases, the creation of an oracle may simply be too hard.

*Correspondence to: Department of Computer and Information Science, University of Pennsylvania, Philadelphia PA 19103. Email: cdmurphy@cis.upenn.edu

These applications typically 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." [2] The absence of a test oracle clearly presents a challenge when it comes to detecting subtle errors, faults, defects or anomalies in software in these domains.

As these types of programs become more and more prevalent in various aspects of everyday life, the dependability of software in these domains takes on increasing importance. Machine learning and scientific computing software may be used for critical tasks such as helping doctors perform a medical diagnosis [3] or enabling weather forecasters to more accurately predict the paths of hurricanes [4]; hospitals may use simulation software to understand the impact of resource allocation on the time patients spend in the emergency room [5]. Clearly, a software defect in any of these domains can cause great inconvenience or even physical harm if not detected in a timely manner.

Without a test oracle, it is impossible to know in general what the expected output should be for a given input, but it may be possible to predict how changes to the input should effect changes in the output, and thus identify expected relations among a set of inputs and among the set of their respective outputs. This approach, introduced by Chen et al., is known as "metamorphic testing" [6]. In metamorphic testing, if test case input x produces an output $f(x)$, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation function t, which can then be applied to the input to produce $t(x)$; this transformation then allows us to predict the expected output $f(t(x))$, based on the (already known) value of $f(x)$. If the new output is as expected, it is not necessarily right, but any violation of the property indicates that one (or both) of the outputs is wrong. This essentially creates a "pseudo-oracle" [7]: though it may not be possible to know whether an output is correct, we can at least tell whether an output is *incorrect*.

Although metamorphic testing has been presented as a simple approach to software testing, to date there is no out-of-the-box automated testing framework to help testers apply the technique in practice. In the same way that JUnit [8] provides a set of core functionality so that testers need not reinvent the wheel each time they perform unit testing, there is a need for a framework that automates the tasks performed by testers when they conduct metamorphic testing. Moreover, unlike JUnit, in which the testers still need to produce test code, a metamorphic testing framework would allow the testers to merely specify the metamorphic properties to be used in testing via a simple notation, with the option of writing more complex test code as necessary.

The need for automation of the metamorphic testing process is particularly clear in the cases in which slight variations in the outputs appear to indicate violations of metamorphic properties, but are not actually the result of errors in the code. This may come about due to imprecisions in floating point calculations that arise during additional executions of the code under test, and may lead to false positives (thinking that there is a defect when there actually is not). In such cases, the results need to be compared to within some threshold, or checked for semantic similarity. Non-deterministic applications also present a challenge, in that it may be necessary to run the program multiple times to get a variety of test outputs, which may then need to be inspected and analyzed in order to construct a profile of the outputs, look for outliers, calculate the distribution or range, etc. Clearly these cases all would require tool support. To date, there has been very little work in investigating the application of metamorphic testing to non-deterministic applications, and yet many applications without test oracles in domains such as machine learning and discrete event simulation may rely on randomization, making them even more difficult to test.

In this paper, we investigate the automation of the metamorphic testing process, and demonstrate that metamorphic testing advances the state of the art in detecting defects in applications without test oracles. This paper makes the following contributions:

1. We introduce the model and architecture of a testing framework called *Amsterdam*, which automates system-level metamorphic testing by treating the application as a black box and checking that the metamorphic properties of the entire application hold after execution. This simplifies the process of conducting metamorphic testing, but also allows for metamorphic

testing to be conducted in the production environment without affecting the user, so that real-world input can be used to drive the test cases.

2. We describe a technique called *Heuristic Metamorphic Testing*, in which the results of multiple invocations are compared according to a domain-specific heuristic to ensure that outputs that are expected to be similar (but not exactly the same) are actually "close enough". This automated technique can be used to avoid false positives that come about from imprecisions in floating point calculations, in which outputs are considered erroneous even though no defect actually exists. It can also be used when applying metamorphic testing to applications that use non-determinism. Although we have previously introduced the Amsterdam framework and Heuristic Metamorphic Testing technique elsewhere [9], this paper describes them in significantly more detail.
3. We provide for the first time the results of new empirical studies conducted on real-world programs (both deterministic and non-deterministic) that have no test oracles to demonstrate the effectiveness of metamorphic testing techniques. These studies show that metamorphic testing is more effective than approaches such as runtime assertion checking and partial oracles.

The rest of this paper is organized as follows. We begin with some background on metamorphic testing in Section 2, then in Section 3 we further motivate the need for automation of the process by which metamorphic testing is conducted in practice. In Section 4, we describe the design and implementation of the Amsterdam framework for automating system-level metamorphic testing. In Section 5, we discuss the Heuristic Metamorphic Testing approach, and describe how Amsterdam applies it to non-deterministic applications. The results of our empirical studies are detailed in Section 6. We then describe related work in Section 7, discuss possible directions for future work in Section 8, and conclude in Section 9.

2. BACKGROUND

2.1. *Metamorphic Testing*

Metamorphic testing was not originally devised as a solution to testing applications without test oracles. Rather, the concept was presented as a way of generating additional test cases from an existing set, particularly by suggesting that test cases that did not reveal defects did not necessarily have to be considered "failures", since they could be used to generate more test cases that may have fault-revealing power [6]. Metamorphic testing sought to extend the use of the algebraic properties of software [10, 11] to demonstrate that such properties could be used to create additional test cases.

As a simple example, 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. Thus, given one set of numbers, we can create three more sets (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2), and get a total of four test cases; moreover, given the output of only the first test case, we can predict what the other three should be.

It was quickly noted that metamorphic testing could be applied to situations in which there is no test oracle [12, 13]: regardless of the values that are used in the test cases, if the relationships between the inputs and between their respective outputs are not as expected, then a defect must exist in the implementation. That is, even if there is no test oracle to indicate whether the outputs are correct, if the new output is not as expected, and the metamorphic property is sound, then the property is considered violated, and therefore a defect must exist.

As an example of how metamorphic testing can be used for applications without test oracles in the domain of machine learning, anomaly-based network intrusion detection systems build up a model of "normal" behavior based on what has previously been observed. This model may be created, for instance, according to the byte distribution of incoming network payloads (since the byte distribution in worms, viruses, etc. may deviate from that of normal network traffic [14]). 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 exists. Besides machine learning [15], other recent research into metamorphic testing has focused on applying it to specific domains of programs without test oracles such as bioinformatics [16] and network simulation [17].

2.2. *Metamorphic Properties*

An open issue in the research on metamorphic testing is, "how does one know the metamorphic properties of the function or application?" Although others have looked into test case selection in metamorphic testing [18], i.e., choosing the test cases most likely to reveal defects, previous work assumes that the tester or developer will have sufficient knowledge of the system or function under test to identify its metamorphic properties, using application- or domain-specific 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 implement an algorithm and perform calculations, the results of which cannot be known in advance; if they could, the program would not be necessary. In Table I, we categorize different classes of metamorphic properties that are common in mathematical functions. 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 given the change to the input.

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	Take the inverse of each element in a set
inclusive	Add a new element to a set
exclusive	Remove an element from a set
compositional	Create a set from some number of smaller sets

Table I. Classes of metamorphic properties

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*) + *n***c*.
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 we take the inverse of each element in *A*, i.e., multiply each element by -1, in order to create an array *A'*, then *Sum*(*A'*) should equal *Sum*(*A*) * -1.
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*.

7. **Compositional**: If the array is decomposed into two smaller arrays *A'* and *A"*, then *Sum*(*A*) should equal *Sum*(*A'*) + *Sum*(*A"*).

These are admittedly very trivial examples and do not fall under the category of programs without test oracles, 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. Such functions are good candidates for metamorphic testing because they are essentially mathematical, and demonstrate well-known properties such as distributivity and transitivity [19].

It is also the case that entire *applications* exhibit such properties, particularly in the domains of interest, such as machine learning [15], discrete event simulation, and optimization. These applications and their metamorphic properties are discussed in our empirical studies below.

3. MOTIVATION

Although it has been demonstrated that it is possible to use metamorphic testing to reveal previously-unknown defects in applications without test oracles [15, 20], the process by which the testing is conducted can benefit from automation, so that testers can be more productive by reusing existing testing frameworks and toolsets, and by running test cases in parallel whenever possible.

3.1. The Need for Automation

Without tool support, the manual transformation of the input data can be laborious and error-prone, especially when the input consists of large tables of data, rather than just scalars or small sets. Machine learning applications, for example, can take input files of thousands or tens of thousands of rows of data; anything but the simplest transformations would need to be automated.

On a similar note, input data that is not human-readable (for instance, binary files representing network traffic) certainly require tools for transformation. For instance, a network intrusion detection system may take as input a collection of network packets represented as streams of bytes, the format of which must adhere to the strict TCP/IP conventions. Any transformation of this input would need to ensure that the new data set contains syntactically and semantically correct packets; however, it is quite difficult to know which parts of the byte stream to modify without tool support.

Additionally, the manual comparison of the program outputs can also cause problems. As with the input data, many applications for which metamorphic testing is appropriate produce large sets of output, and comparing them manually can be error-prone and tedious. In fact, Weyuker includes applications that produce a large amount of output in her definition of "non-testable programs" [2].

Another issue involves metamorphic properties for which changes to the output are to be expected. For instance, in the field of machine learning supervised classifiers, if the input values are multiplied by a constant, then it may be the case that the output values should also be multiplied by the same constant. If the output is large and complex, it can be difficult to manually compare the two outputs to make sure that they are as expected, and obviously a tool like "diff" is not sufficient because the outputs are not expected to be exactly the same.

In all of these cases, one-off scripts could be created, but testers could clearly benefit from a general framework that addresses different types of input transformations and output comparisons for purposes of metamorphic testing. Such a framework would be even more beneficial if it did not require testers to actually write test code; rather, they would only need to use a simple notation to specify the metamorphic properties of the application under test, and let the framework do the rest. More importantly, the testers' productivity could further be increased if the framework automatically handled running multiple invocations of the program in parallel, so that the tester need not wait the entire duration of an additional program run to know the results of the test.

3.2. *Imprecision*

The problem with comparing program outputs - regardless of how it is done - is exacerbated by the fact that imprecisions in floating point operations in digital computers could cause outputs to appear to deviate, even if the calculation is actually correct programmatically. Although errors due to floating point imprecision are not the types of defects that we have set out to discover, comparing the outputs based on an expectation of equality may lead to a false positive, i.e., thinking that there is a defect when there actually is not [21].

Consider the simple case of the sine function, and the metamorphic property sin(α) = sin(α + 2π). In practice, a defect-free implementation may cause a failure of the metamorphic test, due to imprecision in floating point calculations and the representation of π. For instance, the Math.sin function in Java computes the sine of 6.02 radians and the sine of (6.02 + 2 * Math.PI) radians as having a difference on the order of 10^{-15}, which in most applications is probably negligible, but is not exactly the same when checking for equality; thus, a metamorphic property based on checking that the results are equal would lead to a false positive. It may be of interest to the tester to know that the property was violated, of course, but this violation may not actually indicate a defect in the implementation.

3.3. *Non-Determinism*

Non-deterministic programs may yield outputs that are *expected* to deviate between executions, and thus it may be impossible in practice to know whether the output is one that is predicted according to the metamorphic property, given the non-determinism. For example, ranking algorithms in machine learning take sets of data and try to order the elements according to some learned relationship. Such algorithms may rely on randomness, for instance to permute the order of the input data set so that initial ordering does not influence the results, and then take the average ranking over a set of runs. Depending on these permutations, though, the final result may not be deterministic. We may expect the outputs to be "similar" to each other, however, and the degree of expected similarity would need to either be specified by the tester or inferred automatically. In either case, tool support is required.

4. AUTOMATED TESTING FRAMEWORK

To automate the process by which system-level metamorphic testing is conducted, we propose that the tester would simply need to follow these steps:

1. **Specify the application's metamorphic properties.** The tester specifies the application's metamorphic properties using a simple syntax or scripting language. This specification should describe how to transform the input, and what the expected change to the output should be (Section 4.1).
2. **Configure the framework.** The testing framework is configured so that it knows where to find the specification of the metamorphic properties, and how to run the program to be tested (Section 4.2).
3. **Conduct system testing.** The framework is invoked using test input data, as specified by the tester. The transformation of the input data, execution of the program, and comparison of the output are all automatically performed by the framework. The tester is notified about any violation of metamorphic properties, indicating that a defect has been found (Section 4.3).

Aside from facilitating metamorphic testing in the development environment, this technique can conceivably also be used to continually test the application as it runs in the deployment environment, as well. This must be done in such a manner that the end user only sees the results of the main (original) execution, and not from any of the others that are only for testing purposes; Section 4.4 investigates this further.

Figure 1 shows the model of an implementation of such an automated approach. Metamorphic properties of the application are specified by the tester and then are applied to the program input.

The original input is fed into the application, which is treated completely as a black box; a modified version of this input data is also produced, according to the metamorphic property. For each property to be checked, the corresponding input data is then fed into a separate invocation of the application, which executes in parallel but in a separate sandbox so that changes to files, screen output, etc. do not affect the other process(es). When the invocations finish, their results are compared according to the specification; if the results are not as expected, a defect has been revealed. Although not reflected in Figure 1, it should be possible to execute more than two invocations of the program in parallel.

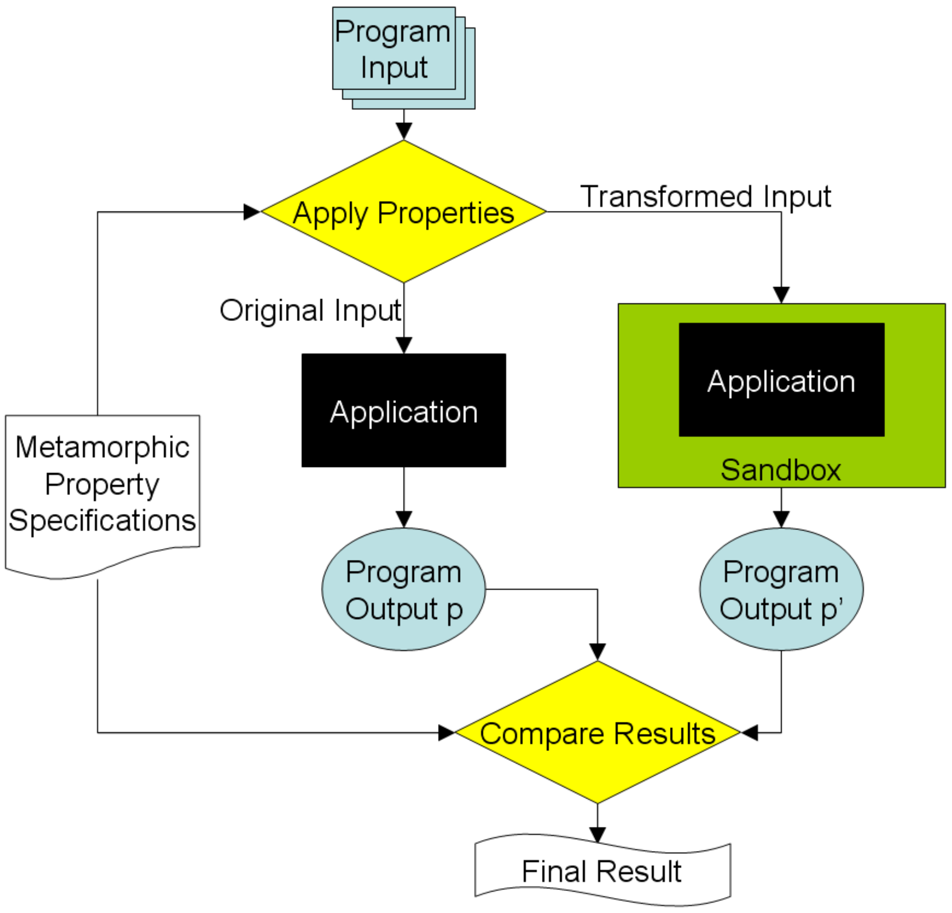

Figure 1. Model of Amsterdam testing framework

Ideally, the tester would not need to write any actual test code per se, nor any test scripts, but rather would only need to specify the metamorphic properties of the application. This can be done by the creator of the algorithm or by the application designer, and does not assume intricate knowledge of (or even access to) the source code or other implementation details.

The rest of this section describes our implementation of a testing framework called *Amsterdam*, which we have introduced previously [9] and discuss in further detail here. The Amsterdam framework enables an application to be treated as a black box so that system-level metamorphic testing can be automated without any modification to the code whatsoever. As described above, multiple invocations of the application are run and their outputs are compared; however, the additional invocations must not affect the original process (or each other) and thus must run in a separate sandbox.

The current implementation of the Amsterdam framework assumes that: the program under test can be invoked from the command line; system input comes from files or from command line arguments; and, output is either written to a file or to standard out (the screen). This may limit the generality of this framework, but according to our preliminary investigations, these assumptions are typically not restrictive in applications in the domains of interest (particularly machine learning, but also discrete event simulation, optimization, etc.). Additionally, when input comes from database tables, mouse clicks, keystrokes, incoming network traffic, etc., an analogous unit testing approach can be used instead, so that metamorphic testing can be conducted at a more granular level [19].

4.1. Specifying Metamorphic Properties

When using the Amsterdam framework, the tester first specifies the metamorphic properties of the application. In our current implementation, this can be done in a text file using a syntax similar to plain English for some simple properties. For instance, if permuting the input to an application is not expected to affect the output (i.e., the resulting output is equal to the output of the initial test case), then the specification would simply be "`if permute (input) then equal (output)`". For more complex properties, an XML file is used for the specification (described below). The examples in this section assume the specifications are written in XML (since the plain-English properties are ultimately pre-processed into XML files), though the ideas and principles will remain the same, regardless of the particular implementation.

The specification of a metamorphic property includes three parts: how to transform the input, how to execute the program (e.g., the command to execute, setting any runtime options, etc.), and how to compare the outputs. Multiple metamorphic properties can either be grouped together in one file, or specified across multiple files.

For **input transformation**, the tester can describe how to modify the entire data set or only certain parts, such as a particular row or column in a table of data. As described above in Section 2.2, we have identified a variety of general categories of metamorphic properties, and the framework supports out-of-the-box input modification functions to match each of these categories: adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing part of the data; adding additional data; and, composing a data set from smaller subsets.

For **program execution**, the tester needs to specify the command used to execute the program. The program is completely treated as a black box, so the particular implementation language does not matter, as long as the program is executable from the command line. Some metamorphic properties may call for different runtime options to be used for the different invocations; those would be specified here.

For **output comparison**, the tester describes what the expected output should be in terms of the original output. In the simplest case, the outputs would be expected to be exactly the same. In other cases, the same transformations described above (adding, multiplying, etc.) for the input may need to be applied to the output before checking for equality. Additionally, the framework also supports checking for *inequality* if a change to the input is expected to cause a change to the output, even if that output cannot be precisely predicted. Last, if the output is non-deterministic, a statistical or heuristic approach can be used, as described in Section 5.

Figure 2 demonstrates an example of a metamorphic property for system testing as specified in an XML file. On line 2, the name of the program to be run is specified, and on lines 3-5 the names of the command-line parameters are given. For the purposes of metamorphic testing, the input is given the placeholder name "training_data" and the output is given the name "model"; other parameters specific to the execution of the program are also specified on line 4. On lines 7 and 10, the types of files for the input and output are given, respectively.

The metamorphic properties to be tested are declared in lines 12-28. On line 13, we say that there is to be a "main" execution, the output of which will be shown to the user. On lines 14-16, we specify a test case called "test1" in which the input is to be permuted. On lines 17-19, we also specify another test case in which the elements of the training data should all be multiplied by 10. Lines 22 and 25 specify how to compare the outputs for the two metamorphic properties: in both cases, the new output (as a result of the test case) is expected to be the same as the original from the "main" execution.

```
<TESTDESCRIPTOR>
   <EXECUTION>/usr/bin/c-marti</EXECUTION>
   <PARAMETERS>
      @input.training_data @output.model --no-permute --no-prob-dist
   </PARAMETERS>
   <INPUT>
      <VAR TYPE="csv_file" NAME="training_data" />
   </INPUT>
   <OUTPUT>
      <VAR TYPE="text_file" NAME="model" />
   </OUTPUT>
   <POST_TEST>
      <BRANCH NAME="main" />
      <BRANCH NAME="test1">
         @op_permute(@input.training_data)
      </BRANCH>
      <BRANCH NAME="test2">
         @op_multiply(@input.training_data, 10)
      </BRANCH>
      <PROPERTY>
         <ASSERT>
            @op_equal(@main.output.model, @test1.output.model)
         </ASSERT>
         <ASSERT>
            @op_equal(@main.output.model, @test2.output.model)
         </ASSERT>
      </PROPERTY>
   </POST_TEST>
</TESTDESCRIPTOR>
```

Figure 2. Example of specification of metamorphic property for system-level testing

Note that with minimal modification, the metamorphic properties specified in this XML file could also be applied to any other program that exhibits the same properties.

```
<TESTDESCRIPTOR>
   <EXECUTION>/usr/bin/c-marti</EXECUTION>
   <PARAMETERS>
      @input.training_data @output.model --no-permute --no-prob-dist
   </PARAMETERS>
   <INPUT>
      <VAR TYPE="csv_file" NAME="training_data" />
   </INPUT>
   <OUTPUT>
      <VAR TYPE="text_file" NAME="model" />
   </OUTPUT>
   <FUNCTION NAME="switch" CLASS="MyOperators" METHOD="reverseDirection"
            TYPE="transform"/>
   <POST_TEST>
      <BRANCH NAME="main" />
      <BRANCH NAME="test1">
         @op_multiply(@input.training_data, -1)
      </BRANCH>
      <PROPERTY>
         <ASSERT>
            @op_equal(@main.output.model, @op_switch(@test1.output.model))
         </ASSERT>
      </PROPERTY>
   </POST_TEST>
</TESTDESCRIPTOR>
```

Figure 3. Example of specification of metamorphic property for system-level testing, with extended functionality

If the framework does not support a specific transformation or comparison feature as required by the tester, functionality can be added by creating a separate component that can be invoked by the framework, according to a specific programming interface (currently implemented in Java). This would also allow the tester to automate the transformation of other input formats not currently supported by the tool, or to compare other output formats.

For instance, Figure 3 shows the specification of another metamorphic property. In this one, multiplying all of the values in the input data by -1 (as specified on line 17) would not produce the exact same output, but rather a similar output but with slight differences. The Amsterdam framework does not have a built-in function for comparing the outputs in such a manner, so the user needs to extend its functionality by creating a new component.

On line 12, the user introduces a new function called “switch”. This function name is mapped to a method called “reverseDirection” in a class called “MyOperators”, which the tester would have implemented. The type of the function is specified on line 13 as a “transform” function, meaning that the implementing method takes one argument, which is the data to be transformed. On line 21, when the outputs are to be compared, Amsterdam will invoke the method mapped to the name “switch”, which produces a new output that can then be checked against the original.

Alternatively, the tester could have created a comparison function that takes the two files to be compared and returns a boolean indicating whether or not they are as expected. In either case, it is easy to add functionality to the Amsterdam testing framework, and such functions can conceivably be reused for testing other applications as well.

4.2. *Configuration*

After specifying the metamorphic properties, the tester then configures the Amsterdam framework to indicate the name of the XML file containing the specifications. Additionally, the tester can also specify how the multiple invocations of the program should be executed. Typically, all of the invocations would be run in parallel, and the outputs compared at the end of all executions, in order to speed up the testing process. However, parallelism may be disabled if the tester wants to also measure resource utilization (memory, CPU, etc.) during a single execution of the application. Also, if the tests are run on a single core/processor, running the applications in parallel will actually cause *more* overhead because of context switching, so the invocations can be run sequentially if so desired (see Section 4.5 for more discussion of performance overhead).

Another reason to run the tests sequentially is if there needs to be some “build up” or “tear down” actions before or after each invocation is run. This may be the case, for instance, if the tests require the creation of a temporary database that will be modified during the program execution. The user can specify “build up” or “tear down” scripts (which are assumed to be executable from the command-line) in the specification of the metamorphic properties.

Sequential execution may also be necessary to support metamorphic properties such as “$ShortestPath(a, b) = ShortestPath(a, c) + ShortestPath(c, b)$ where c is some point in the path from a to b”, i.e., properties that depend on the result of the initial execution of the program in order to conduct the subsequent ones.

The configuration also includes declaring what action to take if a metamorphic test fails. The tester can be notified that the test revealed unexpected behavior through an entry in Amsterdam’s execution log file and/or by a pop-up window.

4.3. *Execution of Tests*

When the application is executed, the testing framework first invokes the original application with the command line arguments provided in the property specification, so that the startup delay of the framework is minimal from the user’s perspective. While the application is running, Amsterdam then makes copies of the input files to use for the additional invocations of the program, then applies the specified transformations to the input files. This is done after invoking the original application because copying and modifying large files can take a long time, and there is no need for the original application to wait. The framework provides out-of-the-box support for the transformation of four different file formats: XML, comma-separated value (CSV), an attribute/value pair format for "sparse" data, and the attribute-relation file format (ARFF). These file formats are commonly used in the domains of interest. Other file formats can be supported by building custom transformation components, as described above.

The framework then starts additional invocations with the newly-generated inputs. When the test processes are to be run in parallel with the main process, the sandbox for each process is provided by simply creating copies of all the input files used by the test processes, and by redirecting screen output to a file so that the user does not see the results of the additional invocations of the program. To make the sandbox more robust, we have also integrated Amsterdam with a virtualization layer called a "pod" (PrOcess Domain) [22], which creates a virtual environment in which the process has its own view of the file system and process ID space and thus does not affect any other processes or any shared files.

At this time, the framework sandbox does not include external entities such as the network or databases, so that modifications made by a test may end up affecting the external state of the original application. Although this appears to limit the usefulness of the tool, we note, however, that in our experiments presented below in Section 6, the current sandbox implementation was sufficient for the applications we tested: none of the applications used an external database or 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 †) 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 [23, 24]. This is left as future work.

Once all processes are complete, the output files are then compared according to the specification of the metamorphic properties. If the output files are not as expected, then a defect has been detected, and the appropriate action can be taken according to the configuration.

4.4. Testing in the Deployment Environment

The framework is also designed to be used in the production environment by the system's end users, so that metamorphic testing can continue even after deployment. Such "perpetual testing" [25] approaches have been shown to be effective at revealing defects that may not have been found before the software is deployed [26]. This addresses one of the practical issues of metamorphic testing, which is "where does the test input come from?"

If the test invocations are run in parallel with the main one, the end user ideally would not even know that the testing was being conducted. In such cases, the results of the tests can be sent back to the development team for use in regression testing and program evolution [27]. The system administrator can specify the following configurations:

- whether to send results of failed test cases or all test cases
- how frequently to send results (i.e., after how many test cases are executed)
- the email address(es) to which the results should be sent
- for failed tests, whether to send the entire input and output files, since sending the entire input file may give rise to privacy issues, and the file may simply be too big to send anyway
- also for failed tests, the metamorphic property that was violated will be supplied

To date we have not investigated the mechanisms by which developers could actually use this information when defects are discovered, but we point out that others have previously looked into using similar field failure data to perform debugging and fault localization [28], or for regression testing [29]. We expect that such techniques are applicable in our case, as well, and leave this as future work.

When using Amsterdam to test a program as it runs in the field, it is assumed that the software vendor would ship the application including the testing framework and specification of the metamorphic properties as part of the software distribution. The customer organization using the software would configure the framework, but aside from that would not need to do anything special at all, and end users ideally would not even notice that the tests were running.

†http://msdn.microsoft.com/en-us/library/ms175158.aspx

4.5. *Performance Overhead*

To demonstrate that the Amsterdam framework incurs limited overhead on the application being tested, we conducted performance tests on a quad-core 3GHz CPU with 2GB RAM running Ubuntu 7.10. Our experiments showed that the performance impact on the main application process (the one seen by the user) comes only from the creation of the sandbox, copying the files for the input, waiting for the test processes to complete, and comparing the results: this was measured at about 400ms for a 10MB input file when just copying the files, and 1.1s when using the "pod" virtualization layer. After that, all other test processes execute on separate cores and do not interfere with the original process (assuming that there are fewer test processes than cores, of course). Thus, the tests can be run with minimal performance impact from the tester's perspective. Note that, without automation, if the processes were run in sequence, then the tester would have to wait for each to finish and the overhead on the testing time would be 100% for each metamorphic property.

5. HEURISTIC METAMORPHIC TESTING

In metamorphic testing, false positives can be a problem if there are small deviations in the results of calculations that are expected to yield the same result. Additionally, many applications without test oracles rely on non-determinism, which limits the effectiveness of metamorphic testing since it makes it more difficult to predict the expected outputs. To address this, we describe a technique called *Heuristic Metamorphic Testing*, based on the concept of "heuristic test oracles" [30]. This variant of metamorphic testing, which was introduced previously [9] and is described in further detail here, permits slight differences in the outputs, in a meaningful way according to the application being tested. By setting thresholds and allowing for application-specific definitions of "similarity", we can reduce the number of false positives and address some cases of non-determinism.

5.1. *Reducing False Positives*

In Heuristic Metamorphic Testing, output values that are "similar" are considered equal, where the definition of "similarity" is dependent on the application being tested. For instance, when the output is numeric (as in the sine function example in Section 3.2), a threshold can be set to check that the values are suitably close. As long as the difference between the values is below the threshold, then the outputs are considered sufficiently "similar", and no violation of the metamorphic property is reported.

Consider the situation of the sine function, for which the the metamorphic property $\sin(\alpha) = \sin(\alpha + 2\pi)$ may be violated in practice because of imprecisions in representing π and in calculating the sine value. We did an experiment in which we checked this property for all values of α ranging from 0 to 2π in increments of 0.0001. For the 62,832 values of α, 52,868 would violate the property in Java if it were checked using equality to compare the outputs; this is a false positive rate of 84.1%. For the C implementation, using the constant `M_PI`, all but one of the values of α violate the property. However, if we check the outputs to within a tolerance of 10^{-10}, then no value of α violates the property in either C or Java. Although this is a very simple example, it demonstrates the need in practice to use thresholds when checking metamorphic properties that involve floating point numbers.

More complex cases may call for heuristics to check for semantic similarity. For example, the Support Vector Machines [31] machine learning classification algorithm would be expected to have the metamorphic property that permuting the input values should not affect the output. However, many implementations do not strictly adhere to this metamorphic property because of approximations used in the program's quadratic optimization algorithm, and the outputs will vary slightly if the inputs are permuted [15]. The output values could conceivably be compared using a threshold, but this assumes that the tester knows in advance how close the results should be, and assumes there is an easy and meaningful way to compare the outputs. Alternatively, we would expect the resulting outputs to be semantically similar to each other, in that the classification of previously-unseen data should be the same when each output is applied, despite any slight differences. That is, even if the outputs themselves are not exactly as expected, if they produce the same classification, they can thus be considered semantically equivalent.

The Amsterdam framework supports techniques for considering such heuristics and either setting thresholds in the comparison of outputs, or specifying how to check semantic similarity, with the intent of reducing false positives. However, it has been argued that although the failure of a metamorphic test in such cases may not necessarily indicate a defect per se, it does reflect a deviation from expected behavior (albeit a very slight one), and thus it is useful to warn the user [20]. Therefore, Amsterdam can be configured to generate such a warning if so desired.

5.2. Addressing Non-Determinism

Non-deterministic applications present a particular challenge to metamorphic testing because it may not always be possible to know what the expected change to the output should be given the change to the input, and then check whether the new test output is as predicted. Here, we describe how Heuristic Metamorphic Testing can be applied to non-deterministic applications, starting with a description of a statistics-based approach.

5.2.1. Statistical metamorphic testing. Guderlei and Mayer presented statistical metamorphic testing [32] as a solution to testing non-deterministic applications and functions that do not have test oracles. Consider a trivial example of a function r that takes two inputs x and y and returns a random integer in the range $[x, y]$. And, of course, $r(10x, 10y)$ would return a random integer in $[10x, 10y]$. However, we cannot simply specify the metamorphic property that $r(10x, 10y)$ should equal $10r(x, y)$, because of the randomization that occurs each time r is invoked.

Statistical metamorphic testing points out that, over many executions of $r(x, y)$, its output values will have a statistical mean μ and variance σ. Even if we cannot know whether μ and σ are correct, we would expect that many executions of $r(10x, 10y)$ should produce a mean 10μ and variance 10σ; of course, given a finite number of executions, they may not be *exactly* 10μ and 10σ, but a statistical comparison (such as a Student T-test [33]) can be used. If the results are not statistically similar, then the metamorphic property has been violated.

5.2.2. Limitations of statistical metamorphic testing. Statistical metamorphic testing is limited to outputs that consist of a set of numbers whose statistical values, such as mean and variance, can be calculated. The approach is not necessarily applicable in the applications of interest presented here, for instance where the output could be a set in which the *ordering* is of prime concern, and the *values* of the set elements are not important. For instance, a machine learning ranking application may produce non-deterministic results when there are missing values in the input data set. Because the output is a listing of elements, there is no mean and variance to calculate, since the values themselves remain the same, but the ordering will be slightly different across multiple executions.

In Heuristic Metamorphic Testing, however, the results of a non-deterministic ranking algorithm can still be compared using some basic domain-specific metrics, such as the quality for each ranking (measured using the Area Under the Curve [34], a common metric for comparing machine learning results), the number of differences between the rankings (elements ranked differently), the Manhattan distance (sum of the absolute values of the differences in the rankings), and the Euclidean distance (in N-dimensional space). Another metric is the normalized equivalence (or Spearman Footrule Distance), which explains how similar the rankings are (1 means exactly the same, 0 means completely in the opposite order) [35].

Furthermore, in practice the user of the system may only be concerned with a small number of elements in the ranking, presumably selected from the top of the list. For a parameterized value X, it may be suitable to calculate the quality of only the top $X\%$ of each ranking, or calculate the “correspondence”, which is simply the number of elements that appear in the top $X\%$ of both rankings, divided by the total number of elements. These metrics, along with the other distance metrics described previously, can help decide whether a pair of rankings is similar in the ranges that are most important, and thus can be used even when the result is non-deterministic.

The issue, of course, is knowing just *how* similar the results should be, so that the heuristic can be applied. In Heuristic Metamorphic Testing, this can be inferred by observing multiple executions with the original input, using the chosen heuristic to create a profile describing how they relate to each other, and then checking that the executions with the new input (after applying metamorphic properties) have a comparably similar profile using a statistical significance measure. The more executions that are observed, the more accurate the profile, and thus the more confident the result.

5.2.3. Examples. Consider an example from the machine learning ranking application MartiRank [36]. Like all supervised machine learning algorithms, MartiRank executes in two phases. The first phase (called the *training phase*) analyzes a set of *training data*, which consists of a set of *attribute* values and an associated *label*, or classification. 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 *testing phase*), the model is applied to another, previously-unseen data set (the *testing data*) where the labels are unknown. In MartiRank, the output of this phase is a ranking such that, when the labels become known, it is intended that the highest valued labels are at or near the top of the ranking, with the lowest valued labels at or near the bottom.

If the data set in the second phase contains any missing/unknown values, MartiRank will randomly place the missing elements throughout the sorted list, making the output non-deterministic. However, because the elements with known values should end up in approximately the same spot over multiple executions, we know that the final rankings should always be somewhat similar, and can use a heuristic (such as normalized equivalence) to measure that similarity. We also know that there is a metamorphic property that if we multiply all values in the data set by 10, the final ranking should still be approximately similar over multiple executions. This property can then be used to conduct metamorphic testing: if the rankings are not similar, then a defect must exist.

To validate this approach, we ran a simple experiment in which we created a data set with missing values and ran MartiRank 100 times, and then compared the similarity of the outputs. The average normalized equivalence (i.e., how similar those rankings were to each other) was 0.996 with a standard deviation of 0.011. We then multiplied all elements in the data set by 10, and ran MartiRank 100 more times. With this new data set, the normalized equivalence was 0.989 and the standard deviation was 0.013. Using a Student t-test, we could determine that the difference between the normalized equivalence values is not statistically significant ($p < 0.05$), meaning that the results are “close enough”. That is, the rankings with the values multiplied by 10 were approximately as similar to each other as those from the original data set. However, when we inserted a defect into the sorting code so that the ordering would be incorrect, the normalized equivalence was 0.663 and the standard deviation was 0.055, indicating that there is a statistically significant difference (i.e., the results are not sufficiently similar), thus revealing the error.

Note that just as metamorphic testing is essentially a pseudo-oracle approach, in that “expected results” are not necessarily “correct results”, Heuristic Metamorphic Testing can only indicate defects, not correctness: if the heuristic comparison is not as expected, that indicates that something is amiss; but if the results *are* as expected, a defect may still exist but not be revealed.

6. EMPIRICAL STUDIES

To demonstrate the effectiveness of an automated metamorphic testing approach and determine how well it can detect defects in software without test oracles, we conducted two empirical studies on various real-world programs and compared the effectiveness of metamorphic testing against some of the techniques suggested by Baresi and Young in their 2001 survey paper [37], which still represents the state-of-the-art. In the first study, we investigated a variety of applications without test oracles in domains such as machine learning and optimization; in the second study, we considered non-deterministic programs.

The experiments presented in this section seek to answer the following research questions:

1. Is metamorphic testing more effective than other techniques for detecting defects in applications without test oracles, particularly in the domains of interest?
2. Is Heuristic Metamorphic Testing more effective than other techniques for detecting defects in non-deterministic applications without test oracles?
3. Which metamorphic properties are most effective for revealing defects in the applications of interest?

Although others have conducted similar studies (most notably Hu et al. [38], described further in Section 6.4), the studies here are the first to consider applications in the particular domains of interest (machine learning, discrete event simulation, and optimization), and the first to evaluate the effectiveness of metamorphic testing when applied to non-deterministic programs.

6.1. *Techniques Investigated*

This section describes the testing techniques employed in the experiments, in addition to metamorphic testing.

6.1.1. Partial Oracle. As previously discussed, it is impossible to know whether the output of these programs is correct for *arbitrary* input, because there is no general test oracle to cover all cases. However, in some trivial cases, it may be possible to know what the correct output should be, based on analysis of the algorithm and understanding how it should perform under certain conditions. A "partial oracle" [39] is one that will indicate correctness or incorrectness of the output program that does not have a test oracle in the general case, but only for a limited set of test cases.

Determining the boundaries of such partial oracles (i.e., at what point does the program cease to have an oracle) is outside the scope of this work, but we assume for each of the applications in our studies that such oracles exist, either because the correct output can easily be calculated by hand, or there is an agreed-upon correct output for the given input within the particular domain.

Weyuker points out that *"Experience tells us that it is frequently the 'complicated' cases that are most error-prone"* [2], thus limiting the effectiveness of the partial oracle. However, we assume that many developers of applications without test oracles would use such a technique, and as it is relatively simple and cheap to implement, we choose to include it in our studies.

6.1.2. Assertion Checking. The use of assertions for fault detection has been employed since the early days of software development [40], and modern programming languages such as ANNA [41] and Eiffel [42], 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. For instance, assertion checking may be used for program safety, such as to ensure that pointers are not null, array indices are in bounds, or that data structure integrity is maintained.

When used in applications without test oracles, assertions can ensure some degree of correctness by checking that function input and output values are within a specified range, the relationships between variables are maintained, and a function's effects on the application state are as expected. As a simple example, in a function to calculate the standard deviation of a set of numbers, the assertions may be that the return value of the function is always non-negative, the values in the input array never change, the size of the input array never changes, etc. While these statements alone do not ensure correctness (many other functions would have the same properties), any violation of them at runtime indicates a defect.

To aid in the creation of assertions, we used the Daikon invariant detection tool [43]. Daikon observes the execution of multiple program runs and creates a set of likely invariants, which can then be used as assertions for subsequent runs of the program. This has been shown to be effective at detecting defects like the ones we use in our studies [44]. Although it is possible to customize the types of invariants that Daikon can detect, in our experiments we only use its out-of-the-box features. Note that the invariants created by Daikon only include function pre- and post-conditions, and do not incorporate any assertions that are within the function itself; future work could consider the effectiveness of runtime assertion checking when using in-function invariants, though these would need to be generated by hand, as discussed in the Threats to Validity section below (Section 6.5).

6.1.3. Approaches Not Investigated. Formal specification languages like Z [45] or Alloy [46] could be used to declare the 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*" [37], that is, the language may not be suitable for describing 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 [47], the specifications need to be complete in order to be of practical use in the general case [48]. Incomplete specifications can be used as partial oracles, though the extent to which this can be controlled in an experiment may introduce additional threats to validity. Thus, we do not consider formal specification languages in our evaluation.

Additionally, it may be possible to perform trace or log file analysis to determine whether or not the program 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 [49], 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. Therefore, we do not consider log file analysis in this study either.

Last, a popular approach to testing such applications is to use a "pseudo-oracle" [7], in which multiple independently-developed implementations of an algorithm process an input and the results are compared: if the results are not the same, then at least one of the implementations contains a defect. Despite the common use of the pseudo-oracle approach, and even though different implementations existed for many of the applications we considered, we could not use the approach in our study because none of the pseudo-oracles consistently produced the exact same output as the versions that we used in the experiments. In some cases, this was by design; in others, it was because of different interpretations of the algorithm. We note, though, that criticism of the pseudo-oracle approach (and N-version programming) is well-documented from both a practical [50] and theoretical [51] point of view, further justifying its omission.

6.2. Study #1: Programs without Test Oracles

In this study, we applied the testing techniques to six real-world programs without test oracles. The goal of the study is to measure the effectiveness of each technique, and to understand why certain approaches are more suitable than others for testing such applications.

6.2.1. Test Subjects. Three of the applications used in this study are from the domain of machine learning. In this domain, programs are designed to detect previously unknown properties of data sets, and thus have no test oracle for arbitrary inputs.

The **Support Vector Machines** (SVM) algorithm [31] is a classification algorithm used in numerous real-world applications, ranging from facial recognition to computational biology.[‡] We tested the SVM implementation in the Weka 3.5.8 toolkit for machine learning [52], written in Java. **C4.5** [53] is another classification algorithm, which uses decision trees. In our experiment, we tested C4.5 release 8, implemented in C. Last, **MartiRank** [36] is a ranking algorithm that is used as part of a prototype application for predicting electrical device failures, developed in C by researchers at Columbia University's Center for Computational Learning Systems.

We also tested the simulation tool **JSim** [54], developed by members of the Laboratory for Advanced Software Engineering Research at the University of Massachusetts-Amherst. Discrete event simulators can be used to model real-world processes in which events occur at a particular time and affect the state of the system [55], and do not have a test oracle because the output is not

[‡]http://www.clopinet.com/isabelle/Projects/SVM/applist.html

Application	Comparison	Math	Off-by-one	Total
C4.5	8	15	5	28
MartiRank	20	23	26	69
SVM	30	24	31	85
JSim	0	1	5	6
Lucene	0	4	11	15
gaffitter	15	19	32	66

Table II. Types of mutants used in Study #1.

known in advance. JSim is implemented in Java and has been used in the modeling of the flow of patients through a hospital emergency room [5].

Another application we investigated is **Lucene** [56], an open-source text search engine library that is part of the Apache project and is implemented in Java. When Lucene produces the results of a search query, each item is given a relevance score, and the results are sorted accordingly. With a known, finite set of documents to search, it is trivial to check whether the items in the query result really do contain the query keywords (and that documents not in the result do not contain the keywords), but the relevance scores and the sorted ordering cannot be known in advance (if they could, there would be no need to have the tool).

The final application investigated in this study is **gaffitter** [57], which is an open-source application implemented in C++ that uses a genetic algorithm to arrange an input list of files and directories into volumes of a certain size capacity, such as a CD or DVD, in a way that the total unused (wasted) space is minimized. This is an example of optimization software, specifically for solving the "bin-packing problem," which is known to be NP-hard.[§] Solutions to optimization problems in the domain of NP-hard qualify as programs without test oracles: after all, by definition, verifying whether a solution is correct is just as hard as finding the solution in the first place.

6.2.2. Methodology. In this experiment, we used mutation testing to systematically insert defects into the source code and then determined whether or not the mutants could be killed (i.e., whether the defects could be detected) using each 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 [58]. These mutations fell into three categories: (1) comparison operators were mutated, e.g., "less than" was switched to "greater than or equal"; (2) mathematical operators were mutated, 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, 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).

In our experiments, we did not consider mutations that yielded a fatal error (crash), an infinite loop, or an output that was clearly wrong (for instance, that violated the output syntax or simply was blank), since any of the considered approaches or even simple inspection would be able to detect such defects.

Table II shows the number and type of mutants that were suitable for our experiment for each of the six applications. The variation in the number of mutants for each application is due to the different sizes of the source code, the different number of points in which mutations could be inserted, and the different number of mutants that were excluded because of fatal errors or obviously wrong output.

6.2.3. Creating Partial Oracles. As the SVM and C4.5 algorithms are popular in the machine learning community, there exist sample data sets for which the correct output is actually known;

[§]http://en.wikipedia.org/wiki/Bin_packing_problem these are used by developers of SVM and C4.5 implementations to make sure there are no obvious defects in their code. Of course, producing the correct output for these inputs does not ensure that the implementation is error-free, but these data sets can be used as simple "partial oracles". We used the "iris" and "golf" data sets from the UC-Irvine machine learning repository [59] for both SVM and C4.5, since the data sets are relatively small in size and the correct output can easily be calculated by hand. We created four variants of each data set, using different values or different ordering of inputs, so that the output would still be predictable, for a total of eight data sets.

For MartiRank, we hand-crafted two data sets for which we could predict what the expected output should be, and confirmed our expectation using the "gold standard" implementation (with no mutants), which we assumed to be error-free. As with the SVM and C4.5 data sets, we created four variants of each one, each of which would yield a predictable output, to produce eight data sets in total.

For JSim, the partial oracle was based on test cases that were created by the developers of the program. These include simple processes that exercise only particular parts of the simulator, such as executing steps in sequential order.

For the Lucene partial oracle, we manually created a simple document corpus in which the first file consisted of only the word "one", the second contained "one two", the third contained "one two three", and so on. Thus, we could easily predict not only the results of queries like "one AND three" but also know the expected relevance ranking. Note that we could not predict the relevance scores themselves, only their relative values, as the scores do not have any meaning on their own.¶

To create a partial oracle for gaffitter, we generated a set of files, each of which had a size that was a power of two (i.e., one byte, two bytes, four bytes, eight bytes, sixteen bytes, etc.). For any given target size, there would be exactly one optimal set of files to reach that value. As an example, if the target were 11 bytes, then a collection including the files of size one, two, and eight bytes would be the only optimal solution. Thus, it would be possible to know what the correct output should be.

Note these data sets for the partial oracle were selected so that the percentage of line coverage was approximately the same for the different applications and the different testing approaches, thus removing any bias due to the number of data sets used. That is, we do not expect that the partial oracles would be significantly more effective if additional data sets were used, assuming the percentage of line coverage did not likewise increase significantly.

6.2.4. Conducting Metamorphic Testing. For the three machine learning applications (SVM, C4.5, and MartiRank), we devised metamorphic properties using the classification described above in Section 2.2. Each property was verified with the default version to make sure that the property was, in fact, expected to hold. The properties are defined in Table III.

(1) Permuting the order of the examples in the input data should create a semantically-equivalent output.
(2) Multiplying each attribute value in the input data by a positive constant (in our case, ten) should create a semantically-equivalent output.
(3) Adding a positive constant (in our case, ten) to each attribute value in the input data should create a semantically-equivalent output.
(4) Negating each attribute value in the input data should create a semantically-equivalent output, but with all values negated.

Table III. Metamorphic properties used for testing machine learning applications

For the metamorphic testing of SVM and C4.5, we used six data sets from the UC-Irvine machine learning repository [59]. These data sets could not be used for MartiRank because it requires

¶http://article.gmane.org/gmane.comp.jakarta.lucene.user/12076

Property	**Description**
romeo OR juliet == juliet OR romeo	Commutative property of boolean operator
romeo juliet^0.25 == romeo^4 juliet	Boosting one term by a factor of 4 should be the same as boosting the other term by a factor of 0.25
romeo OR foo == romeo	Inserting an OR term that does not exist should not affect the results
romeo NOT foo == romeo	Excluding term that does not exist should not affect the results

Table IV. Metamorphic Properties Used for Testing Lucene

numeric input, so we used 10 randomly-generated data sets that covered a variety of equivalence classes, using a technique called "parameterized random testing" [60], to try to obtain line coverage similar to the data sets used in the partial oracle experiment.

For JSim, we identified two metamorphic properties. First, because the event timings as specified in the configuration have no time units, multiplying each by a constant value should effectively increase the overall time to complete the process by the same factor. For instance, if there are N events in the process, and the time to complete each step is t_0, t_1, t_2... $t_{N\text{-}1}$ respectively, and the total time to complete the process is observed to be T, then changing the event timings to $10t_0$, $10t_1$, $10t_2$... $10t_{N\text{-}1}$ would be expected to change the overall process time to $10T$. Second, we considered the effect that changing the event timings would have on the utilization rates for the different resources in the simulation. Specifically, if the event timings were increased by a positive constant, then the utilization rate (i.e., the time the resource is used, divided by the overall process time) of the most utilized resource would be expected to decrease, since the total increase in the overall process time would outweigh the small increase in the resource's time spent working. In the metamorphic testing of JSim, we used the data sets from the emergency room simulation work presented by Raunak et al. [5]

The metamorphic properties for Lucene are based on those presented by Zhou et al. [61], in which the authors applied metamorphic testing to Internet search engines such as Yahoo!, Google, and Microsoft Live Search. We also included others based on the specific syntactic features of Lucene, as described in its user manual. The properties are listed in Table IV. The corpus used for Lucene was the text of Shakespeare's "Romeo and Juliet", separated into five acts and 26 scenes.‖

For gaffitter, we were able to identify two metamorphic properties. First, if the number of generations used in the genetic algorithm were to increase, the overall quality of the result (in this case, the amount of unwasted space) should be non-decreasing. The intuition is that subsequent generations should never reduce the quality, but rather only candidate solutions with better quality should be included. Of course, we cannot say that increasing the number of generations will increase the quality, since an optimal solution may have been reached, but there should not be any decrease in quality. Second, if the sizes of the files were all multiplied by a constant, and the target size ("bin size") were also multiplied by the same constant, the results should not change, since there is no notion of units in the algorithm.

In our metamorphic testing experiment with gaffitter, we used parameterized random testing [60] to generate a collection of 84 files ranging in size from 118 bytes to 14.9MB, and set targets of 1kB, 1MB, and 100MB. The non-deterministic aspects of the genetic algorithm were controlled by specifying a seed for the random number generator.

For all applications, for each data set, each mutated version was executed with both the original and the data set modified according to each metamorphic property, and the results were compared: if the outputs were not as expected, then the mutant was considered to be "killed" (i.e., the defect was found).

‖http://shakespeare.mit.edu/romeo_juliet/index.html

6.2.5. Detecting Invariants. To create the set of invariants that we could use for runtime assertion checking, we applied Daikon to the "gold standard" (i.e., with no mutations) of each application. We executed the applications with each the different data sets described above. This was done in order to reduce the number of "spurious invariants" that Daikon might produce. For instance, if a program is only run once with an input called "input.data", Daikon will detect that the variable representing the input file is always equal to "input.data" and create an invariant as such; clearly this is spurious because it only happens to be true in this one particular case.

Once the set of invariants had been created, we executed each mutated version with the same data sets used for metamorphic testing and used Daikon's invariant checker tool to ensure that none of the invariants were being violated. If a violation occurred, then the defect had been found.

Note that in practice, it may not be possible to derive a complete set of invariants from the "gold standard": after all, if one knew that the implementation were the gold standard, testing it would not be necessary. However, in this experiment, we wanted to create a set of invariants that would most accurately reflect the correctness of the implementation, and this seemed the best way to automate that. An alternative would have been to hand-generate a set of program invariants, but this would have required more detailed understanding of the implementation details. Our aim was to compare the techniques assuming that the tester only had an understanding of what the applications are meant to do, and not how they are implemented.

Additionally, even though we took steps to reduce the number of spurious invariants created by Daikon, some still remained, specifically invariants that asserted that a given variable could only exist in a particular range of values. These invariants were ignored if, upon further analysis, these ranges were deemed to be merely an artifact of the data sets that were used, and not an actual property of the program implementation.

6.2.6. Results. Table V shows the effectiveness (percentage of distinct defects found in each application) for the partial oracle, metamorphic testing, and assertion checking.

Overall, metamorphic testing was most effective, killing 80.6% of the mutants in the six applications. Assertion checking found 69.8% of the 269 defects, and the partial oracle detected 65.4%.

App.	#	Partial Oracle	Metamorphic Testing	Assertion Checking
SVM	85	78 (91.7%)	83 (97.6%)	68 (80.0%)
C4.5	28	22 (78.5%)	26 (92.8%)	28 (100%)
MartiRank	69	47 (68.1%)	69 (100%)	57 (82.6%)
JSim	6	4 (66.7%)	6 (100%)	6 (100%)
Lucene	15	11 (73.3%)	11 (73.3%)	9 (60%)
gaffitter	66	14 (21.2%)	22 (33.3%)	20 (30.3%)
Total	**269**	**176 (65.4%)**	**217 (80.6%)**	**188 (69.8%)**

Table V. Distinct Defects Detected in Study #1.

6.2.7. Analysis. **C4.5.** As opposed to the other applications, runtime assertion checking was more effective than metamorphic testing at revealing the C4.5 defects, although only slightly so (28 mutants killed, compared to 26). Upon investigation, we realized that the mutants not killed by metamorphic testing were both math mutants, in which a calculation was modified. In both cases, this resulted in a violation of an assertion, but not a violation of a metamorphic property.

One of the two defects is found in the code in Figure 4, which comes from the "FormTree" function used to create nodes of the decision tree. Lines 179-182 use a for-loop macro to initialize the values of an array `ClassFreq`, which is used to calculate the frequency with which a class (or "label") appears in the input data. Line 183 then uses the macro to iterate values of `i` from `Fp` (the first index of the array) to `Lp` (the last index). On line 185, the `Class` function is used to determine the class of the item from the input data, and `ClassFreq` is updated with the corresponding weight.

If there is a defect on line 185, such that the addition of the weight is changed to subtraction, the values in `ClassFreq` become negative, in violation of the invariant (detected by Daikon) that the values should always be greater than or equal to zero; thus, the defect is detected. However, metamorphic testing does not reveal this defect: the changes made to the input data do not affect the calculation, and even though the resulting output is incorrect, none of the metamorphic transformations cause the properties to be violated.

```
179 ForEach(c, 0, MaxClass)
180 {
181   ClassFreq[c] = 0;
182 }
183 ForEach(i, Fp, Lp)
184 {
185   ClassFreq[ Class(Item[i]) ] += Weight[i];
186 }
```

Figure 4. Sample code in C4.5 implementation. A defect on line 185 causes a violation of the assertion checking, but not of the metamorphic property.

We see from Table II that C4.5 had a proportionally higher number of math mutants, compared to the other applications; as the example in Figure 4 is a math mutant, as well, it may seem that assertion checking is more effective for these types of defects. However, we observed in all applications that metamorphic testing is approximately as effective for all three types of defects (comparison operator mutations, math operator mutations, and off-by-one errors). The takeaway from the C4.5 result is not that one testing technique is necessarily better than another, but rather that there are cases when assertion checking is useful for detecting defects related to the *value* of variables, whereas metamorphic testing addresses *changes* to those values.

MartiRank. All 69 of the defects in MartiRank were detected by metamorphic testing, making it much more effective than the other two approaches, especially the partial oracle. One of the reasons why the partial oracle was so ineffective is that the data sets we hand-crafted were necessarily much smaller than the ones for metamorphic testing and runtime assertion checking: the partial oracle data sets consisted of 10-20 values, whereas the data sets used with the other techniques had 200-20,000. The small size of the data sets was necessary so that we could accurately and reliably calculate what the correct output should be; anything much larger would be too complex to calculate, or would be so trivial that it would not be of much interest.

Although this seems to put the partial oracle at a disadvantage, note that both the partial oracle data sets and the data sets used for metamorphic testing were attaining approximately the same overall line coverage (around 70%) regardless of their size, and we only considered mutants that were on the execution path, to ensure that there was a possibility that they would affect the program output. Upon analysis of the mutations that were missed by the partial oracle, we discovered that often the defects only caused incorrect outputs when the data sets were large, as expected.

For instance, in one case there was an off-by-one defect in which an error in the allocation of memory for an array meant that one of the elements could be overwritten, but this did not occur when the input was "small" because there was no need for the creation of the additional arrays that did the overwriting.

In another case, a defect in the "merge" step of the merge sort algorithm caused the indices that delineated the sublists to be incorrect, making it possible for elements to be sorted in the wrong order. This defect only affected the final output if the incorrectly sorted elements happened to be on the boundary between two parts of the MartiRank output; this never occurred for the smaller partial oracles, but was more likely to occur in the larger data sets used for the other approaches, primarily because MartiRank continued to sort the larger data sets multiple times, whereas MartiRank had already generated the correct output with the small data set after only a few rounds of sorting.

Recall that the definition of a "non-testable program" is not only limited to software for which an oracle does not exist: it can also include software for which the creation of an oracle is too difficult [2], and the partial oracle by its nature tends to address a smaller, simpler part of the input domain.

This does serve to point out, though, that the metamorphic testing and runtime assertion checking techniques can be used with arbitrary input data of any size, thus improving their effectiveness.

SVM. Metamorphic testing was more effective than the other two approaches for SVM, particularly for off-by-one errors: all 31 were found by metamorphic testing, while assertion checking missed seven and the partial oracle missed four.

Figure 5 shows a sample of the SVM code that was tested in this experiment. The array `m_class` identifies the classification, or "label", for each data point in the input. This particular piece of code attempts to segregate the data into those that have a label of 1 (into set `m_I1`), and those that do not (into set `m_I4`). An off-by-one mutant has been inserted into line 524, such that the for loop omits the final element of `m_class`.

```
524 | for (int i = 0; i < m_class.length-1; i++) {
525 |   if (m_class[i] == 1) {
526 |     m_I1.insert(i);
527 |   } else {
528 |     m_I4.insert(i);
529 |   }
530 | }
```

Figure 5. Snippet of code from Weka implementation of SVM, with an off-by-one error in the for-loop condition on line 524

Metamorphic testing detects this defect because, when the examples in the training data are permuted, a different one is omitted in the loop, and thus it is possible that the contents of both `m_I1` and `m_I4` are different (compared to the original invocation) when using the permuted set: if element *k* had been in `m_I1` in the original execution, but ends up being omitted as a result of permutation, and element *k'* is placed into `m_I4`, then both sets would be different. The result is that, when the program later performs "validation" (i.e., using the training data as testing data to determine the accuracy of the model), the element that was omitted is classified differently in each of the two executions; since the results are expected to be the same, the defect is thus revealed.

On the other hand, when using the partial oracle, either `m_I1` is different or `m_I4` is different (but *not* both, since an element is omitted from one but the other stays the same) in the execution with the mutated version. For the small data sets we used, this did not affect the validation results. Larger data sets would possibly be affected by this difference, but would no longer be useful as partial oracles since the correct output would no longer be predictable.

Assertion checking fails to find this defect because the relevant invariants simply deal with sizes of the sets (e.g., "`m_I1.size` > 0") and the fact that `m_class` does not change. Despite the mutant in line 524, these invariants still hold.

gaffitter. For gaffitter, none of the results are particularly impressive, though we do see again that metamorphic testing is the most effective of the three. One of the reasons that gaffitter (and, presumably, most other implementations of genetic algorithms) is fairly insensitive to the mutations that we introduced is that many of the calculations have only a subtle effect on the output of the program. This is actually by design, since the point of a genetic algorithm is to simulate small changes to the candidate solutions with the hopes that they increase the quality of the result, but any one particular change is unlikely to have a dramatic effect. This is why the partial oracle, in particular, performed so poorly.

For instance, consider the pseudo-code in Figure 6, which summarizes lines 212-226 in GeneticAlgorithm.cc and is typical in most genetic algorithms. This function takes two candidate solutions (i.e., sets of items) called CS1 and CS2, and creates a child candidate solution, which contains some items from CS1 and some from CS2. On line 2, a random number is used to determine whether CS1 and CS2 should cross over at all. If so, on line 3, a crossover point (i.e., an index in the list of items) is randomly chosen. In line 4, the first items (up to the crossover point) of CS1 are merged with the last items (from the crossover point to the end) of CS2.

```
1  crossover(CS1, CS2) {
2     if (rand() < crossover_rate) {
3        crossover_point = rand() * length(CS1);
4        Child = CS1[1, crossover_point] +
               CS2[crossover_point + 1, length(CS2)];
5        return Child;
6     }
7     else return null;
8  }
```

Figure 6. Sample code from genetic algorithm

Considering the mutations such as those we inserted into gaffitter, it is clear that many of them will indeed affect the output of this particular function, but are likely to have little or no effect on the quality of the overall solution. For example, in line 3, an off-by-one mutant may change the value of the crossover point, so that an item from CS1 may incorrectly be replaced instead with an item from CS2 (or vice-versa), but including or excluding a single item is unlikely to have a dramatic effect on the overall quality of the result when the number of items is relatively small, as in our partial oracle. The same can be said of potential off-by-one or mathematical mutants on line 4 in which the items from CS1 and CS2 are selected to create the child: if the wrong items are selected, the result of the function will differ from what is "correct", but if the incorrectly-created child candidate solution does not have a quality that is much higher than what the correctly-created child would have, then it will not be selected for inclusion in the next generation, and the fact that there is a defect is moot.

In cases like the partial oracle that we created for gaffitter, in which there is relatively small number of possible combinations of the files to be packed into the bins, an optimal (and correct) solution will eventually be found, even when there is a defect in the implementation. This explains why the partial oracle was so poor at detecting defects.

Note that we were only able to identify two metamorphic properties for this application. This is partly because for many of the inputs to a genetic algorithm, the effect of changing them cannot be known in advance. For instance, one cannot predict the changes to the output based on changing inputs like crossover rate, mutation rate, etc. After all, if it could be known that a particular value would have a positive effect on the overall result quality, then there would be no need to vary that input. The only inputs for which we could predict changes to the outputs were the list of files, the target size, and the number of generations; these were all used in the metamorphic properties we considered. If we were able to identify more, it follows that more mutants may have been killed by further testing.

JSim and Lucene. Because of the small number of mutations used in the experiments for JSim and Lucene, the results may not be statistically significant, but serve to demonstrate that metamorphic testing is at least as effective as the other approaches. This may not be an important finding in and of itself, but combined with the other results, indicates that metamorphic testing is no less effective than the current state of the art at finding defects in a variety of applications without test oracles.

6.3. *Study #2: Non-Determinism*

The challenge of testing applications without test oracles can be compounded by the fact that some applications in these domains can be non-deterministic, in that multiple runs of the program with the same input will not necessarily produce the same output. Note that being non-deterministic does not necessarily imply that there is no oracle: a program that produces an equally distributed random integer between 1 and 10 is non-deterministic, but has an oracle (specifically, that over many executions, the number of times each integer is returned should be the same; and that no single execution should return a result outside the specified range). However, some non-deterministic applications that do not produce numerical output, or for which the distribution or range of results cannot be known in advance, could be considered not to have an oracle, and thus we include some of these applications in our study.

6.3.1. Test Subjects. In our first study described above, all the applications were deterministic, or were executed with runtime settings to force determinism. In our second experiment, we apply the testing approaches to non-deterministic applications to measure the effectiveness of each technique.

The JSim discrete event simulator (described in the previous experiment) can be non-deterministic depending on the system configuration. Specifically, the amount of time each step in the simulation takes to complete may be random over a range, either using an equal distribution or using a "triangle" distribution with an inflection point at a specified mode; thus, the time it takes for the entire process to complete may be non-deterministic.

The MartiRank application described above also can be non-deterministic. In generating its final output (i.e., the ranking of the input data), MartiRank performs sorting on the elements. If the data set contains missing values (which is very likely to happen when using real-world data [15]), MartiRank will randomly place the missing elements throughout the sorted list. Thus, subsequent executions may yield different results, making the output non-deterministic. Note that in our first study, the MartiRank data sets had no missing values, thus the output was deterministic.

6.3.2. Methodology. For JSim, we used mutation testing to insert defects that were related to the non-deterministic parts of the application. To create defects that would affect these parts of JSim, we systematically inserted 19 different off-by-one mutants related to the event timing, so that when the configuration specified a timing range from A to B for a given event, the actual range would be [A+1, B] or [A, B-1] or [A+1, B-1]. We could not use defects that had ranges starting at A-1 or ending at B+1 because of checks that already existed within the code that would notice such out-of-range errors and raise an exception.

For MartiRank, we were able to use mutation testing as described in the first study and inserted a total of 59 defects (18 mathematical operator mutants and 41 off-by-one mutants). Note that these are not the same set of mutants used in Study #1, since different parts of the program are used in this phase of MartiRank.

6.3.3. Creating Partial Oracles. Although the applications being tested are non-deterministic, a partial oracle can exist if it is possible to enumerate all of the possible "correct" outputs. For both MartiRank and JSim, we created simple data sets for which we could know what all the possible outputs would be, and confirmed this by using the gold standard version (without any mutations). Then, we executed each mutated version up to 100 times; if it produced an output that was not one of the possible correct outputs, the defect had been found.

Of course, it is possible that there would be false negatives in that the mutated version may randomly not produce an erroneous output during the 100 executions. However, given that the input was contrived so that there were only 2-3 possible correct outputs, the likelihood of this happening is incredibly small (around 1 in 10^{17}).

6.3.4. Metamorphic Testing Approaches. In this experiment, we used statistical metamorphic testing (SMT), which has been proposed as a technique for testing non-deterministic applications that do not have test oracles [32]. SMT can be applied to programs for which the output is numeric, such as the overall event timing in JSim, and is based on the statistical properties of multiple invocations of the program. That is, rather than consider the output from a single execution, the program is run numerous times so that the statistical mean and variance of the values can be computed. Then, a metamorphic property is applied to the input, the program is again run numerous times, and the new statistical mean and variance are again calculated. If they are not as expected, then a defect has been revealed.

For JSim, we specified non-deterministic event timing over a range [α, β] and ran the simulation 100 times to find the statistical mean μ and variance σ of the overall event timing in the process (i.e., the time to complete all the steps). We then configured the simulator to use a range [10α, 10β], ran 100 more simulations, and expected that the mean would be 10μ and the variance would be 10σ. Of course, they results would not *exactly* meet those expectations, so we used a Student T-test [33] to see if any difference was statistically significant; if so, then the defect was revealed. We validated this approach with the gold standard implementation (i.e., in which we had not inserted any defects), and found that the resulting distributions were not significantly different, with $p < 0.05$.

Heuristic metamorphic testing (HMT), described above in Section 5, is a similar approach that is based on SMT but can be used for non-deterministic applications or functions that produce non-numerical output, such as the ranking of elements in MartiRank. For such applications, certain metamorphic properties can be applied so that the new output is expected to be "similar" to the original. The issue, of course, is knowing just *how* similar they should be. This can be measured by observing multiple executions with the original input, measuring their similarity (according to some heuristic metric depending on the problem domain, as in the "heuristic oracle" approach for general applications [30]), and then checking that the executions with the new input (after applying metamorphic properties) are comparably similar.

In this experiment, we ran 100 executions of the program with a given input, which produced a list of elements (i.e., the ranking of examples in MartiRank). We could then compare the similarly of those lists of elements using the Spearman Footrule Distance [35], which yields a normalized equivalence rating of how similar the lists are, given that they are always expected to contain the same elements. We then permuted the input (by changing the ordering of the examples in the data set for MartiRank), ran 100 executions with the new input, and calculated the normalized equivalence of the new outputs. As with SMT, we again used a Student T-test to compare the results, and we validated this approach using the gold standard implementation.

6.3.5. Detecting Invariants. As in the previous study, we applied Daikon to the applications and had it observe multiple program executions of the "gold standard" implementation so that it could create a list of invariants. The mutated versions of the software were then executed with the same data sets to see whether any invariants were violated. Because of the non-deterministic nature of the applications, each was run 100 times, since some invariants might only be violated occasionally.

6.3.6. Results. Table VI shows the number of defects found by the three techniques in this study.

The metamorphic testing approaches (heuristic for MartiRank, and statistical for JSim) were most effective for both applications, detecting a total of 59 (75.6%) of the 78 total mutants.

Application	**#**	**Partial Oracle**	**Metamorphic Testing**	**Assertion Checking**
MartiRank	59	15 (25.4%)	40 (67.7%)	27 (45.7%)
JSim	19	0 (0%)	19 (100%)	0 (0%)

Table VI. Defects Found in Study #2.

6.3.7. Analysis. **MartiRank.** Heuristic Metamorphic Testing was the most effective technique at detecting the defects in MartiRank. When MartiRank is run repeatedly, the final result of the ranked elements is expected to be more or less the same each time: even though MartiRank places missing values randomly throughout the list, the other elements will stay in about the same place, so the normalized equivalence (i.e., the heuristic used to measure the "sameness" of multiple lists) is high. One of the metamorphic properties is that permuting the order of the elements should result in a similar ranking (since sorting does not depend on original input order). However, if there is a defect such that the known elements of the list are sorted incorrectly, then the resulting lists will *not* be similar to the original (since the known elements are out of place and the normalized equivalence will be lower), and the defect will be revealed.

As we observed in the first study, the MartiRank partial oracle data sets were too small to be effective, as errors with sorting the known values only appeared in the larger data sets; of course, once the data set got to be too large, the results were no longer predictable, and a partial oracle could not be used.

One reason that runtime assertion checking was not as effective as it was for the deterministic aspects of MartiRank is that Daikon only detected 1927 invariants in this study, compared to 3666 in the first one. We attribute this to the non-determinism: over multiple executions, it may not be true in *every* case that, for instance, one variable is always greater than another, and thus Daikon disregards this as a likely invariant. With fewer invariants, it follows that there are likely to be fewer violations.

JSim. Both the partial oracle and assertion checking techniques were unable to detect any of the JSim defects related to event timing because each approach only considers a single execution of the program, or of the function that produces the random number in the specified range. Consider a function that is meant to return a number in the range [A, B], but has a defect so that the range is actually [A, B-1]. No single execution will violate the invariant that "the return value is between A and B", so assertion checking does not reveal this defect. As for the partial oracle, if it is known that the program calls the function 10 times, for instance, then we can know that the total overall timing (i.e., the sum of the random numbers) should be a value in the range [10A, 10B]. Even with the defect, though, this still will be true: no program execution will have a total overall timing outside that range.

However, statistical metamorphic testing *will* detect this defect because over 100 executions of the program (as in our experiment), the mean and variance show a statistically significant difference compared to what is expected; the other approaches necessarily only run the program once, and do not consider the trend of the program over a number of independent executions.

6.4. *Efficiency*

Although these experiments have shown metamorphic testing to be more effective at detecting defects in applications without test oracles, the goal of the Amsterdam testing framework is to improve the efficiency of the testers, so that they can conduct more tests in a shorter amount of time.

As described above in Section 4.5, the overhead for the current implementation of the Amsterdam framework is on the order of a few hundred milliseconds per test case, which we observed in our experiments to be similar to the overhead incurred by runtime assertion checking. Thus, from a tool standpoint, the efficiency of the two approaches appears to be about the same.

However, the question then arises, "does it take less time to find and specify the metamorphic properties than it does to identify program invariants?" We have not yet investigated this ourselves, but Hu et al. [38] have conducted a study that measured the developer effort for deriving metamorphic properties and program invariants. The subjects of the study were 38 graduate students who had no prior knowledge of metamorphic testing or assertion checking, nor of the applications to which they would apply the techniques. After approximately three hours of training on each technique, and some background about the three applications that they would test, the subjects were asked to identify metamorphic properties and program invariants. On average, for each application the subjects were able to identify around the same number of metamorphic properties and program invariants in a span of a few hours. More significantly, their study showed that these metamorphic properties were actually more effective at revealing defects than the program invariants that the subjects identified.

Thus, it can be concluded that with only a few hours of effort, a developer with no knowledge of metamorphic testing - or even much knowledge of the application under test - can identify metamorphic properties and effectively use a tool such as Amsterdam. Moreover, these results indicate that both the efficiency and effectiveness of the tester using metamorphic testing will not be less than that of one using program invariants.

6.5. *Threats to Validity*

The programs that were investigated in these studies may not be representative of all programs without test oracles. However, these were selected so as to demonstrate that metamorphic testing is applicable to a range of application domains that do not have test oracles: specifically, machine learning (C4.5, MartiRank, SVM), discrete event simulation (JSim), information retrieval (Lucene), and optimization (gaffitter). Moreover, within the domain of machine learning, we chose applications from different subdomains, specifically decision tree classifiers (C4.5), linear classifiers (SVM), and ranking (MartiRank). Given the number of test subjects and the breadth of application domains, plus the fact that many of these applications are (or are part of) industrial systems, we feel that the results can safely be generalized.

Another issue is related to the types of defects that were planted in the applications via mutation testing, and whether or not the ability to detect mutants is a fair metric for comparison of testing approaches. As indicated above, though, mutation testing has been shown to be an accurate approximation of real-world defects [58], and is generally accepted as the most objective mechanism for comparing the effectiveness of different testing techniques [62, 63].

Another potential threat to validity involves the data sets that were used in the experiments. For the partial oracles, the data sets were smaller than the data sets used for metamorphic testing and runtime assertion checking. Of course, the data sets were smaller out of necessity because, for larger data sets, the correct output could not easily be predicted, thus they could not be used as partial oracles. Also, as noted, the line coverage of the data sets was approximately the same for all the testing approaches, and variations of the original partial oracle data sets were generated to attain a better mix of values. It is possible that the results may have been different had different data sets been chosen, but with very few exceptions, we did not see any data set being particularly good (or particularly bad) at revealing defects, and the results would not have changed significantly had any data set been omitted.

In these studies, we used Daikon to create the program invariants that would be used in runtime assertion checking. Although some researchers have questioned the usefulness of Daikon-generated invariants compared to those generated by humans [64], Daikon is generally accepted as the state-of-the-art in automatic invariant detection [65]. We chose to use the tool so that we could eliminate any human bias or human error in creating the invariants, which would require a great deal of knowledge of the source code, considering the complexity of the applications we evaluated. Although in practice invariants may typically be generated by hand, we did not feel that such an approach would scale to larger, more complex systems such as the ones investigated here, especially considering the complications of deriving invariants *after* the software had already been developed.

Daikon can generate spurious invariants if there are not enough program executions, but we mitigated this by running the programs multiple times with different inputs, and by ignoring any invariants that were obviously only relevant to the data sets that were being used, and would not be expected to hold in general. Note that even if some of the invariants that detected defects in the study were, in fact, spurious, this would serve to increase the apparent effectiveness of runtime assertion checking by increasing the number of defects it discovered. Given that our results indicate that metamorphic testing is more effective, any experimental error related to spurious Daikon invariants would, in fact, only strengthen that conclusion.

Last, the ability of metamorphic testing to reveal defects is clearly dependent on the selection of metamorphic properties, and the results may have varied had we selected different ones instead. However, we have shown that a basic set of metamorphic properties (described in Section 2.2) can be used even without a particularly strong understanding of the implementation. Using this approach, therefore, we are demonstrating the *minimum* effectiveness of metamorphic testing; the use of application-specific properties may actually reveal even more defects [20].

7. RELATED WORK

7.1. Addressing the Absence of a Test Oracle

Baresi and Young's 2001 survey paper [37] describes common approaches to testing software without a test oracle. Each of the general approaches from that paper are discussed here. Although other researchers have looked into domain-specific techniques for creating test oracles (e.g., for testing GUIs [66] or web applications [67]), we are not aware of any other recent significant advances in testing applications without test oracles, particularly in the domains of interest: machine learning, scientific computing, simulation, and optimization.

7.1.1. Embedded Assertion Languages. Programming languages such as ANNA [41] and Eiffel [42], 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 [40]. In metamorphic testing, 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 relations between sets of inputs and sets of outputs, as we do here. Additionally, the checking of the assertions in those languages is not allowed to have side effects; in metamorphic testing, the tests are allowed to have side effects (in fact they almost certainly will, since the function or program is called again), but these side effects are hidden from the user.

Others have reported on the effectiveness of runtime assertions in general [68, 69], and of the invariants created by Daikon in particular [65, 64]. Although some have specifically considered using invariants and assertion checking in place of test oracles [70], the empirical studies presented in this paper, as well as studies independently conducted by Hu et al. [38], demonstrate that metamorphic testing is more effective overall than runtime assertion checking in detecting defects in programs without test oracles.

Note that the techniques described in this paper do not preclude the use of the embedded assertions. As demonstrated by the experiments above, the combination of metamorphic testing and runtime assertion checking is more effective than either technique alone. Additionally, we suspect in practice that the identification of metamorphic properties (at the system level or function level) would likely occur at the same point in the software development process as the identification of invariants and assertions, making the approaches complementary.

7.1.2. Extrinsic Interface Contracts and Algebraic Specifications. One approach identified in Baresi and Young's paper is the use of extrinsic interface contracts, which are similar to assertions except that they keep the specifications separate from the implementation, rather than embedded within. Examples include languages like ADL [71] or techniques such as using algebraic specifications [10].

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, a))* == *X* in a Stack implementation) [48], whereas metamorphic properties typically describe how the outputs of a *single* function should relate when the input is changed in a particular way. Additionally, algebraic specifications are typically used to formulate axioms that are then employed to create test cases for particular data structures [72], but are not as powerful for system-level testing in general, since the system itself may not have such properties. Although the tests that are generated do have oracles for the individual cases, the approach cannot be used to create general oracles for arbitrary test cases, i.e., for program operations that do not have associated algebraic specifications, or for the entire system. In this sense, algebraic specifications and metamorphic testing could be considered complementary approaches.

The runtime checking of algebraic specifications has been explored by Nunes et al. [73] and by Sankar et al. [48], though these previous works have considered the specification of only small parts of the application, such as data structures, and do not consider properties of functions in general, nor do they investigate mechanisms for using the properties to conduct system testing, as we do here.

7.1.3. Pure Specification Languages. Coppit and Haddox-Schatz [47] have demonstrated that formal specification languages can be effective in acting as test oracles, by converting the specifications into assertions that can be checked at runtime. Our empirical studies, however, have shown that metamorphic testing is more effective than such an approach for non-deterministic applications. For instance, consider the case in which a function should return a random number in the range [a, b], but because of a defect actually returns a number in [a, b-1]. An invariant created from the formal specification would not detect this defect because no single execution violates the property that the result should be in [a, b]. However, statistical metamorphic testing would detect this defect because, over many executions, the observed mean and variance differ from what is expected.

Richardson et al. [74] presented a technique for developing a test oracle (as state-based test cases) from a specification, and others have looked into the generation of test oracles from program documentation [75, 76], on the assumption that it represents a reasonable approximation of the specification of the system. As mentioned earlier, though, the specifications used in these approaches need to be complete in order to be effective [48]. Others have pointed out that specification languages often must make a trade-off between expressiveness and implementability, such that if the language is sufficiently expressive, it becomes hard to write an implementation [77], or to automatically determine whether the software under test is actually adhering to that specification [37]. Further investigation into this phenomenon could determine which, if any, specification languages are most effective from a practical point of view at revealing defects in programs without test oracles.

7.1.4. Trace Checking and Log File Analysis. Last, it may be possible to perform trace or log file analysis to determine whether or not the program 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. Some researchers have investigated the effectiveness of these techniques for specific domains, such as network communication protocols [78] or graphical user interfaces [79]. However, others have demonstrated that the content of the log/trace files must be carefully planned out, often by someone with great knowledge of the application, in order to be of use in the general case [80], especially if it is necessary to learn a new specification or analysis language [81].

Trace checking approaches often assume that a model of correct execution already exists [82]. However, if the model cannot be known in advance, a related approach is to observe numerous program executions that are assumed to be correct for a set of given inputs, and then look for deviations or anomalies in subsequent executions with a different set of inputs. Such techniques are common in the security domain, e.g., intrusion detection based on anomalous sequences or patterns of system calls [83, 84], or virus detection based on anomalous access to the Windows registry [85]. However, these approaches are likely not applicable to the problem of detecting the types of calculation defects investigated in this paper.

7.2. *Metamorphic Testing*

The idea of applying metamorphic testing to situations in which there is no test oracle was first suggested by Chen et al. [12], though that work only presented the idea in principle, considering situations in which there cannot be an oracle for a particular application [86, 87], or in which the oracle is simply absent or difficult to implement [88]. Others have applied metamorphic testing to specific domains such as bioinformatics [16], network simulation [17], machine learning [20], and graphics [32], using domain-specific or application-specific metamorphic properties. In our work, we have attempted to identify *general* classes of metamorphic properties that can be used in many different domains, including simulation and optimization (two areas to which metamorphic testing had not previously been applied).

The study presented here is most similar to that of Hu et al. [38], in which they compare metamorphic testing and runtime assertion checking. However, their study only compares deterministic applications. To our knowledge, we are the first to compare metamorphic testing to other approaches for testing applications that are non-deterministic and do not have a test oracle. Also, in their study the assertions are created manually by graduate students, and not automatically detected with a tool such as Daikon, as we do here. This stems from the fact that the programs used in their experiments are considerably smaller than the applications we investigate; manually generating the assertions would likely have been error-prone in our experiment, given the complexity of the applications. Although this seems to reduce the novelty of our work, we point out that the authors did not consider an approach based on simple inputs (i.e., the partial oracle) in their evaluation; given the apparent ease of such a testing technique, it is an important contribution to empirically show that metamorphic testing can actually reveal more defects, as we have shown here. It is worth mentioning, though, that our results are consistent with theirs in that metamorphic testing is shown to be more effective than assertion checking at revealing defects in programs that do not have test oracles.

Gotleib and Botella coined the term "automated metamorphic testing" [89] to describe how the process can be conducted automatically, but their work focuses 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. That is, for a specified metamorphic property of a given function, they attempt to construct test data that would violate that property. Like most static techniques, this approach has issues related to scalability, as it only is useful in practice for individual functions and not for entire applications, and thus is complementary to the work presented here.

Beydeda [90] first brought up the notion of combining metamorphic testing and self-testing components so that an application can be tested at runtime, as we do in Metamorphic Runtime Checking, but did not investigate an implementation or consider the effectiveness on testing applications without oracles. Our work extends that initial idea by providing implementation details and evidence of feasibility and effectiveness.

Metamorphic testing is in a sense similar to fuzz testing [91] in that inputs are modified during program execution, and then the function or program is re-run with the new inputs. However, in fuzz testing, there is no expected relationship between the original output and the output that comes from the modified input, unlike in metamorphic testing, in which there is an expected relationship between the outputs, and a violation of that relationship is indicative of a defect. In fuzz testing, the goal is to force erroneous behavior (e.g., a program crash), which is more appropriate for detecting security vulnerabilities than for the types of defects investigated in this paper.

Outside of Guderlei and Mayer's work on statistical metamorphic testing [32], we are not aware of any other investigation of using metamorphic testing techniques for testing non-deterministic applications. As noted in the description of Heuristic Metamorphic Testing, statistical metamorphic testing can only be used for applications that produce outputs that have statistical properties, such as mean and variance, whereas Heuristic Metamorphic Testing is applicable to the more general case of non-deterministic applications in which profiles of outputs can expected to be "similar" (according to some domain-specific definition of similarity) across multiple executions.

7.3. *Domain-Specific Testing*

The work we have presented in this paper is particularly applicable to domains in which there is no test oracle, specifically machine learning, simulation, and optimization, but also areas of scientific computing.

7.3.1. Machine Learning Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular [92, 93, 94], 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 [95] and Weka [52] are two of several frameworks that aid machine learning 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 [59]) for the purpose of comparing result quality, i.e., how accurately the algorithms predict, but not for the software engineering sense of testing.

Testing of intrusion detection systems [96, 97], intrusion tolerant systems [98], and other security systems [99] 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. An intrusion detection system with very few or no false alarms could still have bugs that prevent it from detecting many (or any) actual intrusions, making it completely undependable.

7.3.2. Simulation and Optimization Researchers concerned with the verification of simulation software often acknowledge the need for software testing, but typically do not present techniques beyond what is common for all types of software [100, 101, 102], such as test-driven development or using code reviews. Others have focused on using formal specifications [103], as have researchers investigating the verification of optimization software, as in compiler optimizations [104, 105]. However, as described above, the use of formal languages to act as an oracle can be challenging from a practical point of view, given that the specification often needs to be complete in order to be useful. Additionally, the creation of a formal specification can be fairly complex after the software has already been developed, and requires intimate knowledge of the algorithm being implemented. In the studies presented here, however, we showed that even with a basic understanding of the simulation (JSim) and optimization (gaffitter) software, we were able to create metamorphic properties that were more effective than other approaches at finding defects.

7.3.3. Scientific Computing Recent research presented at the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering [106] addressed many of the issues that arise in testing applications in domains without test oracles. Hannay et al. pointed out that scientific computing software is often developed by scientists (as opposed to professional programmers or software engineers) who understand that software testing is important, but generally feel that they have insufficient understanding of software testing concepts [107]. Heroux and Willenbring suggested a test-driven approach for developing such software [108]; as pointed out above, the identification of metamorphic properties early in the specification phase would likely enhance the effectiveness of metamorphic testing. Hook and Kelly described a testing approach for scientific software that, aside from algorithm verification and scientific validation, calls for "code scrutinization", i.e., specifically looking for common defects such as off-by-one errors or incorrect use of array indices [109]. As demonstrated by our empirical studies, metamorphic testing is also effective at detecting such errors, and may be used in conjunction with code inspections.

8. FUTURE WORK

Potential areas for future work include:

- **Supporting automatic fault localization.** Although the approaches can record a test failure and know which test failed, it may not be obvious what is the root cause of the failure (invalid system state, invalid function arguments, configuration, environment, combination thereof, etc.). Thus, the system could take a snapshot of the relevant parts of the state (i.e., the ones that could have affected the outcome of the test) and record those in the failure log as well, for further analysis. If these logs are aggregated by the software developer, a failure analysis technique (e.g., [110] or [111]) could be used to try to isolate the fault. However, given that research into fault localization techniques is typically targeted at defects that cause the application to crash, or assume the presence of an oracle, challenges will arise in determining the source of the error in applications for which the correct output cannot be known in advance.
- **Automatically detecting properties that can be used to aid in the generation of tests.** Although prior work has been done in automatically determining algebraic specifications [112] and in categorizing metamorphic properties [15], as of now it is necessary for the software developer or tester to discover and specify the properties and/or write the tests required for metamorphic testing. It may be possible to automate this process, using static or dynamic techniques.
- **Exploring the soundness of metamorphic properties.** The violation of a metamorphic property does not *necessarily* indicate a defect: it is possible that the property may not be sound, i.e., not expected to hold in *every* case. Future work could investigate how a tester would be able to tell the difference. Others have demonstrated that an unsound model (or, in our case, unsound metamorphic properties) may reveal defects that more restrictive sound properties would not [113], even though that raises the risk of false positives. For instance, we previously pointed out a metamorphic property in the ML ranking algorithm MartiRank that permuting the order of the input data should not affect the output, but only assuming that

the values in the input are all distinct, since MartiRank uses stable sorting. However, we can remove this assumption and concede that although this metamorphic property is not sound (because for some inputs, it will not be true), it may reveal actual defects that may not be detected if we included the original constraint that all values must be distinct. That is, there may be a tradeoff of accepting false positives in the hopes of finding more errors.

9. CONCLUSION

In this paper, we have presented an automated approach to metamorphic testing applications that do not have test oracles, and described a testing framework called *Amsterdam*, which addresses some of the practical limitations of metamorphic testing so that it can be an efficient technique for testing applications that deal with large, complex data sets. Amsterdam automates system-level metamorphic testing by treating the application as a black box and checking that the metamorphic properties of the entire application hold after execution. Testers do not need to write any test code, but rather only need to specify the metamorphic properties using a simple notation. Additional executions of the program are run in parallel with the original, in order to make the testing process more efficient.

Additionally, we have introduced a new technique called *Heuristic Metamorphic Testing*, which can be used to avoid false positives that come about from imprecisions in floating point calculations, and can also be used in the testing of non-deterministic applications. In Heuristic Metamorphic Testing, the application is run multiple times to build a profile of its outputs, using a domain-specific heuristic, and then the metamorphic transformation is applied: the application is then run multiple times again, and a measurement is taken to see if the profile of the new outputs is statistically similar to what is expected. If not, then a defect has been detected.

Last, we conducted two empirical studies designed to measure the effectiveness of metamorphic testing at finding defects in the domains of interest. In the first study, we investigated three supervised machine learning classifiers, in addition to applications in the domains of discrete event simulation, information retrieval, and optimization. In the second study, we applied the techniques to non-deterministic applications. Both studies showed that metamorphic testing is more effective than other approaches, specifically using partial oracles or runtime assertion checking.

Addressing the testing of applications without oracles has been identified as a future challenge for the software testing community [114], and we hope that our findings here help others who are also concerned with the quality of such programs.

ACKNOWLEDGEMENT

We would like to thank Adam Cannon, T.Y. Chen, Lori Clarke, Lee Osterweil, Kuang Shen, Sal Stolfo, and Junfeng Yang for their guidance and assistance. The authors are members of the Programming Systems Lab, funded in part by NSF grants CNS-0905246, CNS-0717544, CNS-0627473, and CNS-0426623, and NIH grant 1U54CA121852-01A1.

REFERENCES

1. Young M. Perpetual testing. *Technical Report AFRL-IF-RS-TR-2003-32*, Univ. of Oregon February 2003. URL http://handle.dtic.mil/100.2/ADA412542.
2. Weyuker EJ. On testing non-testable programs. *Computer Journal* November 1982; **25**(4):465–470.
3. Ho JWK, Lin MW, Adelstein S, dos Remedios CG. Customising an antibody leukocyte capture microarray for systemic lupus erythematosus: Beyond biomarker discovery. *Proteomics - Clinic Application* 2010; **in press**.
4. Williams JK, Ahijevych DA, Kessinger CJ, Saxen TR, Steiner M, Dettling S. A machine learning approach to finding weather regimes and skillful predictor combinations for short-term storm forecasting. *Proc. of the Sixth Conference on Artificial Intelligence Applications to Environmental Science*, 2008.
5. Raunak M, Osterweil L, Wise A, Clarke L, Henneman P. Simulating patient flow through an emergency department using process-driven discrete event simulation. *Proc. of the 2009 ICSE Workshop on Software Engineering in Health Care*, 2009; 73–83.
6. Chen TY, Cheung SC, Yiu S. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01*, Dept. of Computer Science, Hong Kong Univ. of Science and Technology 1998.

7. Davis MD, Weyuker EJ. Pseudo-oracles for non-testable programs. *Proc. of the ACM '81 Conference*, 1981; 254–257.
8. JUnit. http://www.junit.org/.
9. Murphy C, Shen K, Kaiser G. Automated system testing of programs without test oracles. *Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA)*, 2009; 189–199.
10. Cody Jr WJ, Waite W. *Software Manual for the Elementary Functions*. Prentice Hall, 1980.
11. Lipton RJ. New directions in testing. *Distributed Computing and Cryptography* 1991; **2**:191–202.
12. Chen TY, Tse TH, Zhou ZQ. Fault-based testing without the need of oracles. *Information and Software Technology* 2002; **44**(15):923–931.
13. Zhou ZQ, Huang DH, Tse TH, Yang Z, Huang H, Chen TY. Metamorphic testing and its applications. *Proc. of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004.
14. Wang K, Stolfo S. Anomalous payload-based network intrusion detection. *Proc. of the Seventh International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
15. Murphy C, Kaiser G, Hu L, Wu L. Properties of machine learning applications for use in metamorphic testing. *Proc. of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2008; 867–872.
16. Chen TY, Ho JWK, Liu H, Xie X. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 2009; **10**(24).
17. Chen TY, Kuo FC, Liu H, Wang S. Conformance testing of network simulators based on metamorphic testing technique. *Lecture Notes in Computer Science* 2009; **5522**.
18. Chen TY, Huang DH, Tse TH, Zhou ZQ. Case studies on the selection of useful relations in metamorphic testing. *Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, 2004; 569–583.
19. Murphy C, Shen K, Kaiser G. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. *Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009; 436–445.
20. Xie X, Ho J, Murphy C, Kaiser G, Xu B, Chen TY. Application of metamorphic testing to supervised classifiers. *Proc. of the 9th International Conference on Quality Software (QSIC)*, 2009.
21. Brilliant S, Knight JC, Leveson NG. The consistent comparison problem in n-version software. *IEEE Transactions on Software Engineering* 1989; **15**:1481–1485.
22. Osman S, Subhraveti D, Su G, Nieh J. The design and implementation of Zap: A system for migrating computing environments. *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002; 361–376.
23. Willmor D, Embury SM. A safe regression test selection technique for database-driven applications. *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM)*, 2005; 421–430.
24. Willmor D, Embury SM. An intensional approach to the specification of test cases for database applications. *Proc. of the 28th International Conference on Software Engineering (ICSE)*, 2006; 102–111.
25. Osterweil L. Perpetually testing software. *Proc. of the The Ninth International Software Quality Week*, 1996.
26. Murphy C, Kaiser G, Chu M, Vo I. Quality assurance of software applications using the in vivo testing approach. *Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009; 111–120.
27. Elbaum S, Hardojo M. An empirical study of profiling strategies for released software and their impact on testing activities. *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004; 65–75.
28. Clause J, Orso A. A technique for enabling and supporting debugging of field failures. *Proc. of the 29th International Conference on Software Engineering (ICSE)*, 2007; 261–270.
29. Orso A, Apiwattanapong T, Harrold MJ. Leveraging field data for impact analysis and regression testing. *Proc. of the 9th European Software Engineering Conference*, 2003; 128–137.
30. Hoffman D. Heuristic test oracles. *Software Testing and Quality Engineering* 1999; :29–32.
31. Vapnik VN. *The Nature of Statistical Learning Theory*. Springer, 1995.
32. Guderlei R, Mayer J. Statistical metamorphic testing - testing programs with random output by means of statistical hypothesis tests and metamorphic testing. *Proc. of the Seventh International Conference on Quality Software*, 2007; 404–409.
33. Gosset WS. The probable error of a mean. *Biometrika* March 1908; **6**(1):1–25.
34. Hanley JA, McNeil BJ. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 1982; **143**:29–36.
35. Spearman C. Footrule for measuring correlation. *British Journal of Psychology* June 1906; **2**:89–108.
36. Gross P, Boulanger A, Arias M, Waltz D, Long PM, Lawson C, Anderson R, Koenig M, Mastrocinque M, Fairechio W, *et al.*. Predicting electricity distribution feeder failures using machine learning susceptibility analysis. *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, 2006.
37. Baresi L, Young M. Test oracles. *Technical Report CIS-TR01 -02*, Dept. of Computer and Information Science, Univ. of Oregon 2001.
38. Hu P, Zhang Z, Chan WK, Tse TH. An empirical comparison between direct and indirect test result checking approaches. *Proc. of the 3rd International Workshop on Software Quality Assurance*, 2006; 6–13.
39. Pezzè M, Young M. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.
40. Clarke LA, Rosenblum DS. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes* May 2006; **31**(3):25–37.
41. Luckham D, Henke FW. An overview of ANNA - a specification language for ADA. *Technical Report CSL-TR-84-265*, Dept. of Computer Science, Stanford Univ. 1984.
42. Meyer B. *Eiffel: The Language*. Prentice Hall, 1992.
43. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely programming invariants to support program evolution. *Proc. of the 21st International Conference on Software Engineering (ICSE)*, 1999; 213–224.

44. Nimmer JW, Ernst MD. Automatic generation of program specifications. *Proc. of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, 2002; 232–242.
45. Abrial JR. *Specification Language* Z. Oxford Univ Press, 1980.
46. Jackson D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(2):256–290.
47. Coppit D, Haddox-Schatz J. On the use of specification-based assertions as test oracles. *Proc. of the 29th Annual IEEE/NASA Software Engineering Workshop*, 2005.
48. Sankar S. Run-time consistency checking of algebraic specifications. *Proc. of the 1991 International Symposium on Software Testing, Analysis, and Verification (TAV)*, 1991; 123–129.
49. Murphy C, Kaiser G. Improving the dependability of machine learning applications. *Technical Report CUCS-49-08*, Dept. of Computer Science, Columbia University 2008.
50. Anderson T, Lee PA. *Fault Tolerance: Principles and Practice*. Prentice Hall, 1981.
51. Knight J, Leveson N. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering* 1986; **12**(1):96–109.
52. Witten IH, Frank E. *Data Mining: Practical Machine Learning Tools and Techniques, 2nd Edition*. Morgan Kaufmann, 2005.
53. Quinlan JR. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.
54. Wise A. JSim agent behavior specification language.
55. Pidd M. *Computer simulation in management science*. Wiley, 1998.
56. Apache Lucene. http://lucene.apache.org.
57. Augusto DA. Genetic algorithm file fitter. http://gaffitter.sourceforge.net/.
58. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proc. of the 27th International Conference on Software Engineering (ICSE)*, 2005; 402–411.
59. Newman DJ, Hettich S, Blake CL, Merz CJ. UCI repository of machine learning databases. University of California, Dept of Information and Computer Science 1998.
60. Murphy C, Kaiser G, Arias M. Parameterizing random test data according to equivalence classes. *Proc. of the 2nd International Workshop on Random Testing*, 2007; 38–41.
61. Zhou ZQ, Tse TH, Kuo FC, Chen TY. Automated functional testing of web search engines in the absence of an oracle. *Technical Report TR-2007-06*, Dept. of Computer Science, Hong Kong University 2007.
62. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* Sept 2006; **32**(9):733–752.
63. Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. *Proc. of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, 2009; 69–80.
64. Polikarpova N, Ciupa I, Meyer B. A comparative study of programmer-written and automatically inferred contracts. *Proc. of the 2009 International Symposium on Software Testing and Analysis (ISSTA)*, 2009; 93–104.
65. Nimmer JW, Ernst MD. Invariant inference for static checking: An empirical evaluation. *Proc. of the 10th International Symposium on the Foundations of Software Engineering (FSE)*, 2002; 11–20.
66. Memon A, Banerjee I, Nagarajan A. What test oracle should I use for effective GUI testing? *Proc. of the 18th International Conference on Automated Software Engineering (ASE)*, 2003; 164–173.
67. Sprenkle S, Pollock L, Esquivel H, Hazelwood B, Ecott S. Automated oracle comparators for testing web applications. *Proc. of the 18th International Symposium on Software Reliability*, 2007; 117–126.
68. Leveson NG, Cha SS, Knight JC, Shimeall TJ. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering* April 1990; **16**(4):432–443.
69. Rosenblum DS. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* January 1995; **21**(1):19–31.
70. Cheon Y. Abstraction in assertion-based test oracles. *Proc. of Seventh International Conference on Quality Software*, 2007; 410–414.
71. Sankar S, Hayes R. Adl: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices* August 1994; **29**(8):13–21.
72. Gannon JD, McMullin P, Hamlet RG. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems* July 1981; **3**(3):211–223.
73. Nunes I, Lopes A, Vasconcelos V, Abreu J, Reis LS. Checking the conformance of Java classes against algebraic specifications. *Proc. of the International Conference on Formal Engineering Methods (ICFEM), volume 4260 of LNCS*, Springer-Verlag, 2006; 494–513.
74. Richardson DJ, Aha SL, O'Malley TO. Specification-based test oracles for reactive systems. *Proc. of the 14th International Conference on Software Engineering (ICSE)*, 1992; 105–118.
75. Clarke D, Jéron T, Rusu V, Zinovieva E. Stg: a tool for generating symbolic test programs and oracles from operational specifications. *Proc. of the 8th European software engineering conference*, 2001; 301–302.
76. Peters DK, L PD. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering* March 1998; **24**(3):161–173.
77. Tyugu E, Saabas A. Problems of visual specification languages. *Proc. of Information Technologies in Science, Education, Telecommunication and Business*, 2003; 155–157.
78. Fujiwara S, v Bochmann G, Khendek F, Amalou M, Ghedamsi A. Test selection based on finite state models. *IEEE Transactions on Software Engineering* June 1991; **17**(6):591–603.
79. Memon AM, Pollack ME, Soffa ML. Automated test oracles for GUIs. *Proc. of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE)*, 2000.
80. Andrews JH, Zhang Y. Broad-spectrum studies of log file analysis. *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, 2000; 105–114.
81. Yantzi DJ, Andrews JH. Industrial evaluation of a log file analysis methodology. *Proc. of the 5th International Workshop on Dynamic Analysis*, 2007.

82. Wang X, Wang J, Qi ZC. Automatic generation of run-time test oracles for distributed real-time systems. *Lecture Notes in Computer Science* 2004; **3235/2004**:199–212.
83. Eskin E, Lee W, Stolfo S. Modeling system calls for intrusion detection with dynamic window sizes. *Proc. of DARPA Information Survivabilty Conference and Exposition II (DISCEX)*, 2001.
84. Warrender C, Forrest S, Pearlmutter B. Detecting intrusions using system calls: alternative data models. *Proc. of the 1999 IEEE Symposium on Security and Privacy*, 1999; 133–145.
85. Apap F, Honig A, Hershkop S, Eskin E, Stolfo S. Detecting malicious software by monitoring anomalous windows registry accesses. *Proc. of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002)*, 2001; 16–18.
86. Chen TY, Kuo FC, Tse TH, Zhou ZQ. Metamorphic testing and beyond. *Proc. of the International Workshop on Software Technology and Engineering Practice (STEP)*, 2004; 94–100.
87. Chen TY, Tse TH, Zhou ZQ. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002; 191–195.
88. Chan WK, Cheung SC, Leung KRPH. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research* April-June 2007; **4**(1):60–80.
89. Gotleib A, Botella B. Automated metamorphic testing. *Proc. of 27th Annual International Computer Software and Applications Conference (COMPSAC)*, 2003; 34–40.
90. Beydeda S. Self-metamorphic-testing components. *Proc. of the 30th Annual Computer Science and Applications Conference (COMPSAC)*, 2006; 265–272.
91. Sutton M, Greene A, Amini P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
92. Briand L. Novel applications of machine learning in software testing. *Proc. of the Eighth International Conference on Quality Software*, 2008; 3–10.
93. Cheatham TJ, Yoo JP, Wahl NJ. Software testing: a machine learning experiment. *Proc. of the ACM 23rd Annual Conference on Computer Science*, 1995; 135–141.
94. Zhang D, Tsai JJP. Machine learning and software engineering. *Software Quality Control* June 2003; **11**(2):87–119.
95. Demsar J, Zupan B, Leban G. Orange: From experimental machine learning to interactive data mining. [www.ailab.si/orange], Faculty of Computer and Information Science, University of Ljubljana.
96. Mell P, Hu V, Lippmann R, Haines J, Zissman M. An overview of issues in testing intrusion detection systems. *Technical Report Tech. Report NIST IR 7007*, National Institute of Standard and Technology.
97. Nicholas JP, Zhang K, Chung M, Mukherjee B, Olsson RA. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering* 1996; **22**(10):719–729.
98. Madan B, Goševa-Popstojanova K, Vaidyanathan K, Trivedi KS. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Performance Evaluation Journal* 2004; **56**(1-4):167–186.
99. Balzarotti D, Cova M, Felmetsger V, Jovanovic N, Kirda E, Kruegel C, Vigna G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. *Proc. of the 2008 IEEE Symposium on Security and Privacy*, 2008; 387–401.
100. Balci O. Verification, validation and accreditation of simulation models. *Proc. of the 29th conference on winter simulation*, 1997; 135–141.
101. Kleijnen JPC. Verification and validation of simulation models. *European Journal of Operational Research* April 1995; **82**(1):145–162.
102. Sargent RG. Verification and validation of simulation models. *Proc. of the 37th conference on winter simulation*, 2005; 130–143.
103. Tsai WT, Liu X, Chen Y, Paul R. Simulation verification and validation by dynamic policy enforcement. *Proc. of the 38th annual Symposium on Simulation*, 2005; 91–98.
104. Lacey D, Jones ND, van Wyk E, Frederiksen C. Proving correctness of compiler optimizations by temporal logic. *Proc. of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2002; 283–294.
105. Lerner S, Millstein T, Chambers C. Automatically proving the correctness of compiler optimizations. *Proc. of the ACM SIGPLAN conference on programming language design and implementation*, 2003; 220–231.
106. Carver J. Report from the second international workshop on software engineering for computational science and engineering. *Computing in Science & Engineering* 2009; **11**(6):14–19.
107. Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G. How do scientists develop and use scientific software? *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009; 1–8.
108. Heroux MA, Willenbring JM. Barely sufficient software engineering: 10 practices to improve your CSE software. *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009; 15–21.
109. Hook D, Kelly D. Testing for trustworthiness in scientific software. *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009; 59–64.
110. Baah GK, Gray A, Harrold MJ. On-line anomaly detection of deployed software: a statistical machine learning approach. *Proc. of the 3rd International Workshop on Software Quality Assurance*, 2006; 70–77.
111. Liblit B, Aiken A, Zheng AX, Jordan MI. Bug isolation via remote program sampling. *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003; 141–154.
112. Henkel J, Diwan A. Discovering algebraic specifications from Java classes. *Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.
113. Hallem S, Chelf B, Xie Y, Engler D. A system and language for building system-specific, static analyses. *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002; 69–82.
114. Bertolino A. Software testing research: Achievements, challenges, dreams. *Proc. of ICSE Future of Software Engineering (FOSE)*, 2007; 85–103.