View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

# Metamorphic Runtime Checking of Applications without Test Oracles

Christian Murphy
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
cdmurphy@cis.upenn.edu

Gail Kaiser, Jonathan Bell, Fang-Hsiang Su
Dept. of Computer Science
Columbia University
New York, NY 10027
{kaiser, jbell, mikefhsu}@cs.columbia.edu

## ABSTRACT

Challenges arise in testing applications that do not have test oracles, i.e., for which it is impossible or impractical to know what the correct output should be for general input. Metamorphic testing, introduced by Chen et al., has been shown to be a simple yet effective technique in testing these types of applications: test inputs are transformed in such a way that it is possible to predict the expected change to the output, and if the output resulting from this transformation is not as expected, then a fault must exist.

Here, we improve upon previous work by presenting a new technique called *Metamorphic Runtime Checking*, which automatically conducts metamorphic testing of both the entire application and individual functions during a program's execution. This new approach improves the scope, scale, and sensitivity of metamorphic testing by allowing for the identification of more properties and execution of more tests, and increasing the likelihood of detecting faults not be found by application-level properties alone. We also discuss a technique for automatically discovering functions' metamorphic properties, and present the results of new studies that demonstrate that Metamorphic Runtime Checking advances the state of the art in testing applications without oracles.

## Categories and Subject Descriptors

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

## General Terms

Reliability

## Keywords

Software Testing, Test Oracle, Metamorphic Testing, Quality Assurance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ICSE* 2014 Hyderabad, India
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

## 1. INTRODUCTION

In the testing of software, a "test oracle" [37] is required to indicate whether the output is correct for the given input. Despite a recent interest in the testing community in creating and evaluating test oracles [45], still there are a variety of problem domains for which a practical test oracle does not exist in the general case. Applications in the fields of scientific computing, simulation, machine learning, etc. 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"* [48]. Thus, in the general case, it is not possible to know the correct output in advance for arbitrary input. In other domains, such as optimization, determining whether the output is correct is just as difficult as it is to derive the output in the first place, and creating an efficient, practical oracle may not be feasible.

Although some faults in such programs - such as those that cause the program to crash or produce results that are obviously wrong to someone who knows the domain - are easily found, and partial oracles may exist for a subset of the input domain, subtle errors in performing calculations or in adhering to specifications can be much more difficult to identify without a practical, general oracle.

Much of the recent research into addressing the so-called "oracle problem" has focused on the use of a technique called metamorphic testing [8] in domains such as bioinformatics [9], machine learning [29, 50], scientific computing [21], and simulation [10, 40]. As the name implies, in metamorphic testing changes are made to existing test inputs in such a way (based on the program's "metamorphic properties") that it is possible to predict what the change to the output should be. That is, if program input $I$ produces output $O$, additional test inputs based on transformations of $I$ are generated in such a manner that the change to $O$ (if any) can be predicted. In cases where the correctness of the original output $O$ cannot be determined, i.e., if there is no test oracle, program failure can still be detected if the change to $O$ is not as expected when using the new input.

Through our own past investigations into metamorphic testing [29, 30, 31], we have garnered three key insights. First, the metamorphic properties of individual functions are often different than those of the application as a whole. Thus, by checking for additional and different relationships, we can reveal defects that would not be detected using only the metamorphic properties of the full application. Second,

the metamorphic properties of individual functions can be checked in the course of executing metamorphic tests on the full application. This addresses the problem of generating test cases from which to derive new inputs, since we can simply use those inputs with which the functions happened to be invoked within the full application. Third, when conducting tests of individual functions within the full running application in this manner, checking the metamorphic properties of one function can sometimes detect defects in other functions, which may not have any known metamorphic properties, because the functions share internal application state.

In order to advance the state of the art in testing applications that do not have practical, general test oracles, this paper seeks to make metamorphic testing more effective by improving: the number and types of metamorphic properties that can be checked (**scope**); the number and types of metamorphic tests that are run for a single program input (**scale**); and the likelihood of revealing subtle faults that would not cause a violation of an application-level metamorphic property (**sensitivity**).

In order to realize these improvements, we present a solution based on checking the metamorphic properties of the entire application *and* those of individual functions (or methods, procedures, subroutines, etc.) as the full application runs. That is, the program under test is not treated only as a black box, but rather metamorphic testing also occurs *within* the application, at the function level, in the context of the running program. This will allow for the execution of more tests and also makes it possible to check for subtle faults inside the code that may not cause violations of the application's metamorphic properties.

Although we have previously presented some initial observations regarding the implementation and effectiveness of such a technique [32], this paper extends that work by making four contributions:

1. A new type of testing called **Metamorphic Runtime Checking** (Section 3). This is a new approach that improves metamorphic testing in that metamorphic properties of individual functions, not only the entire application, are also specified and then checked as the program is running.

2. A new technique for automatically discovering metamorphic properties (Section 4). This is done by starting out with a set of possible properties and ruling them out as they are violated during program execution, analogous to the automatic detection of likely program invariants [16].

3. An architecture of a Metamorphic Runtime Checking framework called **Columbus** (Section 5). This architecture allows for checking both application-level and function-level metamorphic properties as the program runs.

4. The results of new studies that show that Metamorphic Runtime Checking is more effective than other techniques in detecting faults in programs without test oracles (Section 6).

Although Metamorphic Runtime Checking does not preclude other testing approaches, and could presumably be applied to applications that *do* have oracles, the focus of this work is on those that do not.

# 2. BACKGROUND

Metamorphic testing [8] was introduced as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any faults, by reusing test cases to create additional test inputs whose expected outputs can be predicted. In metamorphic testing, if input `x` produces an output `f(x)`, the function's (or application's) metamorphic properties can be used to guide the creation of a transformation function `t`, which can be applied to the input to produce `t(x)`; this transformation then allows us to predict the output `f(t(x))`, based on the (already known) value of `f(x)`. In the case where `f` has an oracle, then if we know that `f(x)` is correct, we could also know whether `f(t(x))` is correct. Studies have shown that these additional test cases have fault-finding capabilities beyond that of the original test set [51].

For a simple example of metamorphic testing, 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 should multiplying each value by -1. Furthermore, other transformations should 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.

Clearly metamorphic testing can be very useful in the absence of an oracle: even if it were not possible to know whether the initial output is correct, if the new output that results from transforming the input does not have the expected relationship with the original output, then a fault must exist in the implementation [11, 51]. Although satisfying these metamorphic properties does not indicate correctness, a violation of a property would indicate that one (or both) of the outputs is incorrect.

As an example from the domain of machine learning, anomaly-based network intrusion detection systems build a model of "normal" traffic based on previous observations, and then later look for outliers that may be indicative of an attack. The model may be created according to the byte distribution of incoming network payloads, since the distribution in worms, viruses, etc. may deviate from that of normal network traffic [47]. When a new payload arrives, its byte distribution is then compared to the model, and anything deemed anomalous causes an alert. When testing such a program, it may not be possible to know *a priori* whether a particular input should raise an alert, since it 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, and this property is violated, then a fault must exist in the implementation.

Despite recent successes in demonstrating that metamorphic testing can be effectively applied to real-world programs in domains for which there is no practical, general test oracle (e.g., [9], [10], [40], [50], etc.), there is still room for improvement when it comes to its ability to detect faults. By identifying more metamorphic properties (i.e., increasing the scope), deriving more test cases for each program input (increasing the scale), and checking for subtle faults that may not cause a violation of application-level metamorphic properties (increasing the sensitivity), we can make metamorphic testing even more effective.

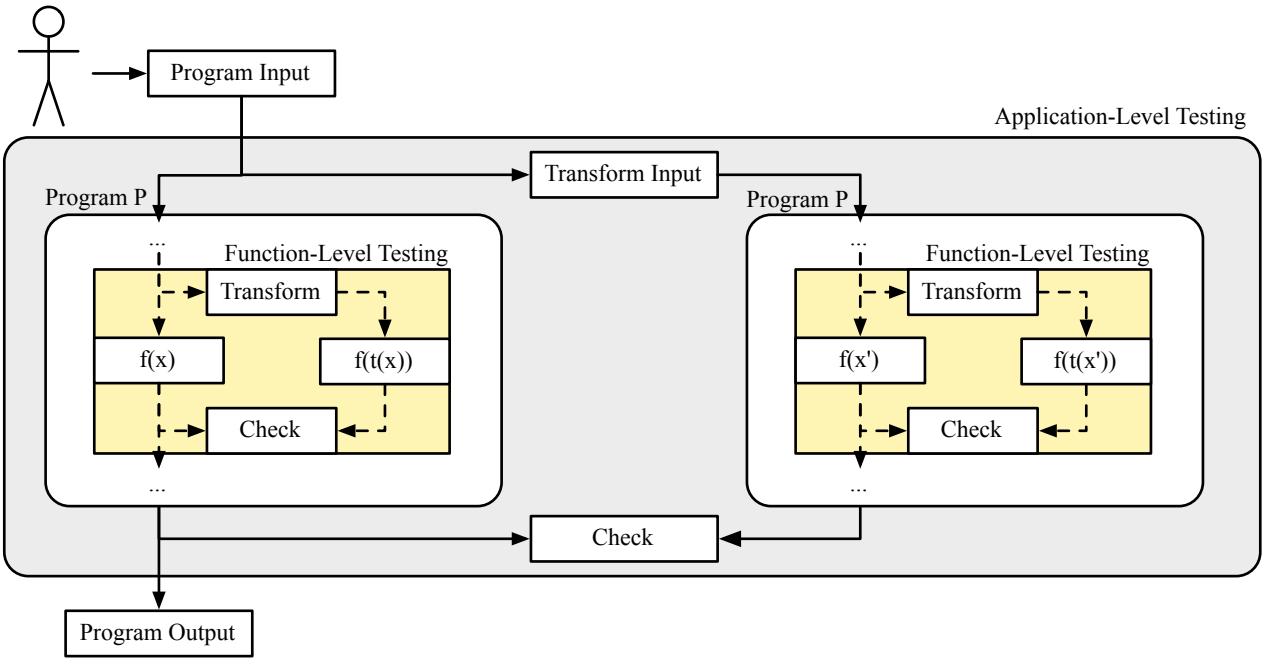


**Figure 1: Model of Metamorphic Runtime Checking of program `P` and its constituent function `f`.**

## 3. APPROACH

Here we introduce a new technique called *Metamorphic Runtime Checking*, which can be considered a cross between metamorphic testing and runtime assertion checking [12], in that the metamorphic properties that are checked are analogous to program invariants: at the point in the program execution in which the function is called, its metamorphic properties are expected to hold, given the current function inputs and application state. Although we cannot know for certain just from observation that a given property will *always* hold, if the metamorphic property is ever violated, then a fault has been revealed.

Metamorphic Runtime Checking is an extension of our previous work in Automated Metamorphic System Testing [31], in which an application's metamorphic properties are specified and then the program is executed multiple times: once with the original input, and then again for each of the transformed inputs, according to the properties, after which the new outputs are compared to the original to determine whether any properties have been violated. Despite the demonstrated effectiveness of this approach, it does not utilize properties of individual functions, only those of the entire application.

In our new approach, additional metamorphic tests are logically attached to the functions for which metamorphic properties have been specified. Upon a function's execution, the corresponding tests are executed as well: the arguments are modified according to the specification of the function's metamorphic properties, the function is run again in the same application state as the original, and the output of the function with the original input is compared to that of the function with the modified input. If the result is not as expected according to the specified property, then a fault has been exposed.

As shown in Figure 1, the tester provides a program input to a Metamorphic Runtime Checking framework, which then transforms it according to the metamorphic property of the program `P` (for simplicity, this diagram only shows one metamorphic property, but a program may, of course, have multiple). The framework then invokes `P` with both the original input and the transformed input; as seen at the bottom of the diagram, when each program invocation is finished, the outputs can be checked according to the property.

While each invocation of `P` is running, though, metamorphic properties of individual functions can be checked as well. As shown on the left side of Figure 1, in the invocation of `P` with the original program input, before a function `f` is called, its input `x` can be transformed according to the function's property to give `t(x)`. The function is called with each input, and then `f(t(x))` is evaluated according to the original value of `f(x)` to see if the property is violated.

Meanwhile, in the additional invocation of `P` (right side of the diagram), function-level metamorphic testing still occurs for function `f`, this time using input `x'`, which results from the transformed program input to `P`. In this case, `f(t(x'))` and `f(x')` are compared.

By incorporating function-level metamorphic properties into the testing, we can improve the scope and scale of metamorphic testing by running more tests using more properties and more inputs. In this example, if we used only the application-level property of `P`, we would run just one test. However, by also considering `P`'s functions, we can now check two properties and run a total of three tests. Additionally, this also allows us to improve the sensitivity of metamorphic testing by revealing subtle faults at the function level that may not violate application-level properties.

In order to conduct Metamorphic Runtime Checking, the tester must first identify the metamorphic properties of the application to be tested and its constituent functions. This

can be done manually, using domain knowledge or guidelines as presented in [29], or can be done automatically, as described in Section 4.

Once the metamorphic properties have been determined, they must be expressed in such a way that the properties can be checked at runtime. In previous work, we discussed the use of configuration files for specifying application-level properties [31], and we have also explored using code annotations based on the Java Modeling Language [23] for function-level properties [32]. Any of these approaches can be used in Metamorphic Runtime Checking.

Last, testing can commence. Unlike in conventional unit testing, in Metamorphic Runtime Checking the tester need not construct any specific test harness; rather, metamorphic testing is conducted by simply executing the program with selected test inputs. The testing framework would allow for the checking of both application-level and function-level properties, and report any violations. The architecture for such a framework is presented in Section 5.

# 4. AUTOMATIC DISCOVERY OF LIKELY METAMORPHIC PROPERTIES

An open issue in the research on metamorphic testing is, "where do the metamorphic properties come from?" Previous attempts to answer this question involve the use of guidelines [29, 30], constructing properties as combinations of existing ones [24], and using machine learning to suggest properties based on structural similarity to other pieces of code [22]. Here we present a new approach to discovering properties based on observing program execution.

As described above, a metamorphic property of a function `f` describes a transformation of the input such that we can predict the change to the output. That is, we apply a transformation function `t` to the input `x` to get `t(x)`, and then calculate `f(t(x))`, which we then compare to some transformation `g` of the original output `f(x)`. Thus, we can define a metamorphic property of a function `f` as "a pair of functions (`t`, `g`) such that `f(t(x))` = `g(f(x))` for all `x`". It follows, then, that to determine the metamorphic properties of `f`, we need to find functions `t` and `g` for which this is true.

In practice, of course, it is not feasible to find *all* functions `t` and `g`, but in the same way that invariant detection tools start with a set of possible invariants and eliminate/refine them as they are violated during program execution [16], we can discover metamorphic properties of the code by trying combinations of functions `t` on the input and functions `g` on the output. Any combination for which `f(t(x))` = `g(f(x))` is never violated can therefore be hypothesized as a likely metamorphic property.

In order to assist with the identification of metamorphic properties, a testing framework that implements Metamorphic Runtime Checking can have a "discovery mode": rather than checking for the violation of a user-specified set of properties, it can try combinations of transformation functions on the input and output domains, and report as likely properties those combinations that were never violated. Note that these are not the same as program invariants, which relate to relationships between variables; these properties relate to relationships between multiple function executions.

As an example, consider a function `k` whose input is an array of ints `A`. For this particular input domain, we can apply well-known set transformations (e.g., permuting, sorting, reversing) as well as numeric transformations on the individual elements (e.g., adding a constant, multiplying by a constant, etc.) to get new inputs `A'`. The simplest metamorphic property is that changing the input does not change the output, i.e., `k(A')` = `k(A)`. For each transformation that derives a new input from `A`, if this property is violated, then it is omitted from consideration. Subsequent invocations of `k` with different inputs can refine this set of properties even further, until the only ones remaining are those that were never violated.

Given the output domain of `k`, we can try additional transformations as well by looking for changes to `k(A)` such that it equals `k(A')`. Overall, if we have $m$ possible transformations in the input domain and $n$ in the output domain, we can try all $mn$ combinations; those that are never violated are thus reported as likely properties.

There are, of course, well-known limitations to such an approach. Observing program execution to derive properties may lead to false positives (i.e., thinking that a property is sound when it actually is not) or false negatives (when what should be a sound property is not reported because of a bug in the implementation), and only reports what the properties *are* and not what they *should be*.

However, the Metamorphic Runtime Checking approach described in Section 3 is not restricted to only those properties discovered by this technique. The tester is certainly free to identify and specify additional properties, or to rule out spurious properties that only hold for the selected inputs; we simply suggest this technique simply to assist in identifying likely properties from a list of known common ones for various input and output domains.

Due to space limitations, we do not investigate this approach further in this paper, in terms of the soundness of the properties and their fault-finding capabilities, except to point out that in our empirical studies (Section 6), the function-level properties discovered by this approach are effective at revealing bugs in the software not identified by application-level properties, and thus improve the overall effectiveness of metamorphic testing.

# 5. ARCHITECTURE

In this section we introduce the architecture of a testing framework for Metamorphic Runtime Checking, which we call *Columbus*.

## 5.1 Overview of Application-Level Testing

The Columbus framework is an extension of the Amsterdam framework for application-level metamorphic testing, which we introduced in our work on Automated Metamorphic System Testing [31] and is only summarized here.

In the Amsterdam framework, metamorphic properties of the entire application are specified using a configuration file. The framework executes the program with the original input, and then modifies the program inputs according to the properties. The program is then run again with the new inputs, but additional invocations are run in separate sandboxes so that they do not interfere with each other. Once all executions have finished, the outputs are checked according to the specified properties, and any violations are reported.

The Columbus framework builds upon Amsterdam by allowing testers to also specify function-level properties, and then by conducting additional metamorphic testing as the program invocations execute, as described below.

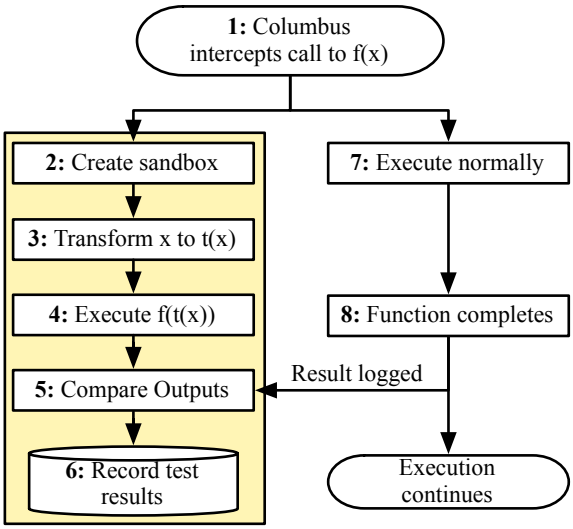


**Figure 2: Model of Columbus framework for function-level metamorphic testing.**

## 5.2 Function-Level Testing

In order to perform metamorphic testing of an individual function during the execution of the program, the Columbus framework needs to: execute the function with the original input and get the original output; transform the input according to the specified metamorphic property; call the function again, this time with the new input, but in the same application state as the original function call; and then get the new output and compare it to the original, again according to the specified property, to see whether the property was violated. All but the first of these steps would be repeated for each of the function's specified properties.

One challenge is that the testing framework must ensure that the additional function invocations do not adversely affect the state of the running program: the goal is that the instrumented application should still produce the same output as an uninstrumented version, even though Metamorphic Runtime Checking is being conducted. For instance, assume that the program calls function `f` with input `x`, which is transformed according to the metamorphic property to `t(x)`. In order for the program to continue to run as normal after the metamorphic tests have been run, the testing framework must ensure that the return value sent to the calling function is `f(x)`, and not `f(t(x))`. Likewise, any side effect of calling `f(x)`, such as modifying a global variable, must be permitted, but the framework must make sure that any side effect of calling `f(t(x))` is hidden.

Since existing runtime assertion checking techniques (as surveyed in [12]) and monitoring tools (such as Gamma [36]) do not support multiple function executions and do not safeguard against side effects, a new solution is required.

Figure 2 shows how the Columbus framework supports function-level metamorphic testing. First, the framework intercepts the call to function `f`, which can be accomplished through wrapping the function or by injecting code. Note that neither the signature of the function nor its return type is changed, and the calling function is never aware that metamorphic testing is being conducted.

Before the body of function `f` is invoked, the framework must create a "sandbox" so that the extra function calls do not affect the state of the running application. In native languages like C and C++, this can be accomplished by forking the process: the test inherits the state of the program at the time of the original function call, but does not affect it when the function is run a second time. In managed languages like Java and C#, for which forking the entire virtual machine is unwieldy, deep cloning can be used to make copies of all objects that may be affected. A new thread can then be spawned here, using references to the sandboxed/cloned objects, instead of the original.

In the process or thread running the test, the input `x` to the function `f` is first modified according to the particular metamorphic property to give `t(x)`. The transformation that should be applied to the input can be specified using code annotations [32], a configuration file, etc. Next the (uninstrumented) implementation of the function is invoked using this new input `t(x)`.

Meanwhile, in the process/thread that is doing the original function call (right-hand side of Figure 2), the function `f` is invoked with the unmodified input `x`, and once it completes, the output `f(x)` can be logged or immediately shared with the test. If the tests are running in a separate process from the original, then a mechanism like shared memory or message passing would need to be used here, otherwise this can be done with a global/static variable.

Once the test case has determined that the original function call has finished, e.g., using signals between processes or by using shared flags, the outputs `f(x)` and `f(t(x))` are compared according to the metamorphic property. As with the transformation, the function `g` used for checking the outputs must be specified, and then success or failure can be indicated here by determining whether `f(t(x)) = g(f(x))`.

Finally, the test terminates, the output `f(x)` that results from the original input is returned to the calling function, and the rest of the program continues as normal.

## 5.3 Prototype Implementations

In order to conduct the experiments described in the following section, we created prototype implementations of the Columbus framework for both C and Java applications. For specifying the metamorphic properties and generating the test cases, we built upon our previous work [32] using code annotations based on the Java Modeling Language [23]. For creating the sandbox and sharing the function outputs, we modified the Invite framework used for "in vivo testing" [28], which allows for the execution of unit tests alongside a running program. The performance overhead introduced by these tools is discussed in Section 6.4.

# 6. EMPIRICAL STUDIES

To evaluate the effectiveness of Metamorphic Runtime Checking at detecting faults in applications without test oracles, we compare it to runtime assertion checking using program invariants. 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 [34]. While satisfying the invariants does not ensure correctness, any violation of them at runtime indicates an error.

Table 1: Applications used in experiment.

| Name | Application Domain | Language | LOC | Functions | Invariants | Application-Level Metamorphic Properties | Function-Level Metamorphic Properties |
|---|---|---|---|---|---|---|---|
| C4.5 | classification | C | 5,285 | 141 | 27,603 | 4 | 40 |
| GAFFitter | optimization | C++ | 1,159 | 19 | 744 | 2 | 11 |
| JSim | simulation | Java | 3,024 | 468 | 306 | 2 | 12 |
| K-means | clustering | Java | 717 | 46 | 137 | 4 | 12 |
| LDA | topic modeling | Java | 1,630 | 103 | 1,323 | 4 | 28 |
| Lucene | information retrieval | Java | 661 | 57 | 456 | 4 | 26 |
| MartiRank | ranking | C | 804 | 19 | 3,647 | 4 | 15 |
| PAYL | anomaly detection | Java | 4,199 | 164 | 19,730 | 2 | 40 |
| SVM | classification | Java | 1,213 | 49 | 2,182 | 4 | 4 |

Although we are not aware of any other work that has specifically attempted to determine the "state of the art" for testing applications without oracles, runtime assertion checking is mentioned as a common and effective technique in the literature, e.g., Baresi and Young's survey paper [4] on techniques for testing applications without test oracles, and Kanewala and Bieman's paper [21] on testing programs used in computational science. Additionally, using invariant detection tools, this approach is easy to automate, and does not rely on a human tester, as do other approaches such as N-Version Programming or creating a formal specification.

## 6.1 Experimental Setup

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

1. Is Metamorphic Runtime Checking more effective than using runtime assertion checking for detecting faults in applications without test oracles?
2. What contribution do application-level and function-level metamorphic properties make to the effectiveness of Metamorphic Runtime Checking?
3. Is Metamorphic Runtime Checking suitable for practical use?

### 6.1.1 Target Applications

In these experiments, we applied both runtime assertion checking and Metamorphic Runtime Checking to nine real-world applications that are representative of different domains that have no practical, general test oracles: supervised machine learning (classification, clustering, and ranking), data mining (information retrieval and topic modeling), discrete event simulation, and optimization. The applications include: **C4.5** [39], a decision tree-based machine learning classification algorithm; **GAFFitter** [17], an optimization program that uses a genetic algorithm implemented to solve the bin-packing problem; **JSim** [41], a discrete event simulation engine developed at UMass Amherst; **K-means** [25], a clustering algorithm as implemented in the Mahout [3] framework; **Latent Dirichlet Allocation (LDA)** [7], a topic modeling algorithm, also as implemented in Mahout; **Lucene** [2], a text search engine library that is part of the Apache framework; **MartiRank** [18], a machine learning ranking algorithm, which was developed by researchers at Columbia University's Center for Computational Learning Systems; **PAYL** [47], an anomaly-based intrusion detection system implemented by researchers in Columbia University's Intrusion Detection System Lab; and **Support Vector Machines (SVM)** [46], a machine learning classification algorithm, as implemented in the Weka [49] open-source toolkit.

More detail about the applications is provided in Table 1. Note that for the Java applications, the information only includes the implementations of the specific algorithms, and not any underlying frameworks.

### 6.1.2 Identifying Program Invariants

To identify assertions, we used the Daikon invariant detection tool [16]. 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. 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.

To create the set of invariants that we could use for runtime assertion checking, we applied Daikon to each application, using the following data sets as input: for SVM, C4.5, and K-means, 10 of the most popular data sets from the UC-Irvine repository of machine learning data sets [33]; for MartiRank, ten randomly generated data sets of various sizes; for PAYL, network traffic on our department's LAN over a one-hour period; for Lucene and LDA, the corpus of Shakespeare's comedies [27]; and for GAFFitter, a collection of 84 files ranging in size from 118 bytes to 14.9MB.

### 6.1.3 Identifying Metamorphic Properties

To identify the application-level metamorphic properties for the experiment, we used our implementations of the Columbus framework in "discovery" mode as described above in Section 4, based on observing program executions with the same data sets used for detecting invariants with Daikon. Note that, for these properties, we simply treat the application as a black box, apply common transformations to the inputs (based on the input domain and file format), apply

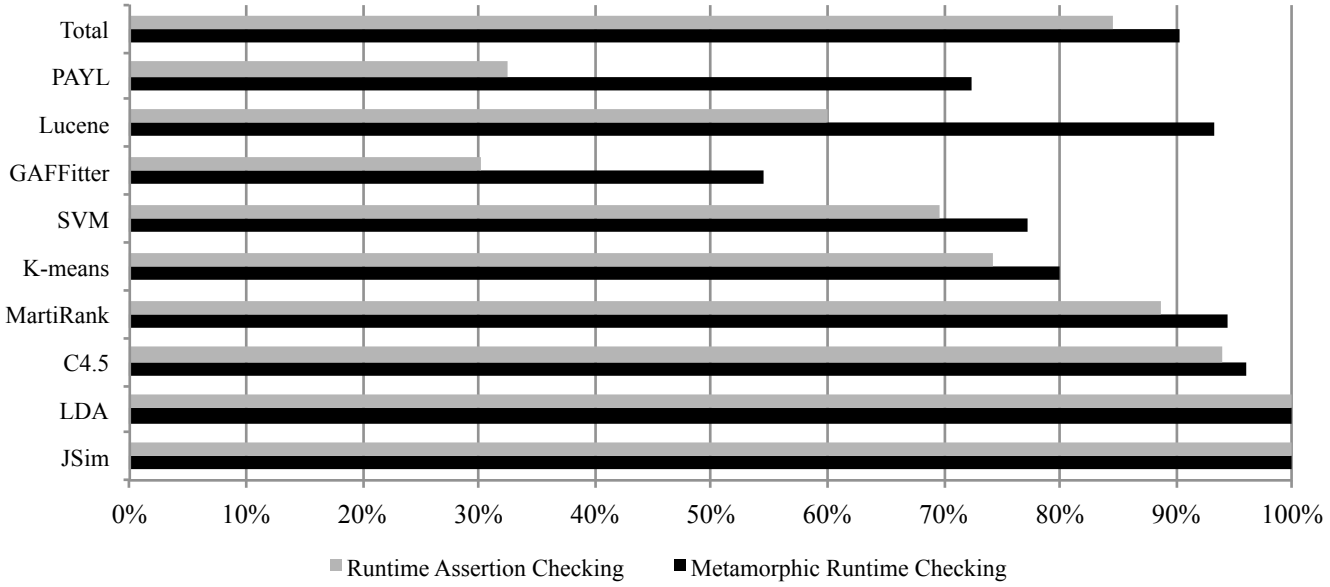


**Figure 3: Results of mutation analysis, showing percentage of mutants killed by each approach.**

common transformations to the outputs, and look for any combinations that are never violated.

To identify function-level properties, we again used our Columbus implementations in "discovery" mode, this time for the individual functions, using the same program inputs as described above. Note that we did not construct separate inputs for the functions, but rather the framework used whatever values were passed along during the execution of the entire program.

Since our Columbus implementations can currently only discover properties based on function arguments and return values, and not on side effects or application state, this meant that we could not identify properties for functions that took no inputs and/or had no return value. And since we were not involved in the development of any of these applications, someone with more knowledge of the domains or implementations probably could have identified additional properties. However, as shown below, the properties that were discovered still had fault-finding capabilities, and previous studies [20, 26] have shown that these are the types of properties that tend to be identified by humans anyway.

The total numbers of application-level and function-level metamorphic properties are listed in Table 1.

### *6.1.4 Methodology*

To determine the effectiveness of the testing techniques, we used mutation analysis to systematically insert faults into the source code of the applications described above, and then determined whether the mutants could be killed (i.e., whether the faults could be detected) using each approach. Mutation analysis has been shown to be suitable for evaluation of effectiveness [1], and is generally accepted as the most objective mechanism for comparing the effectiveness of different testing techniques [15, 44].

Each mutated version of the program had exactly one mutation. Mutations that yielded a fatal runtime error (crash), an infinite loop, or an output that was clearly wrong (for instance, not conforming to the expected output syntax, or simply being blank) were discarded since any reasonable approach would detect such faults. We also did not consider "equivalent mutants" for which all inputs produced the same program output as the original, unmutated version, e.g., those mutants that were not on the execution path for any test case.

For each mutated version, we conducted runtime assertion checking with the invariants detected by Daikon. If any invariant were violated, the mutant was considered killed. We then performed Metamorphic Runtime Checking on the same mutated versions to determine whether any of the metamorphic properties discovered by Columbus were violated. The inputs used for mutation analysis were the same as those used for detecting invariants and properties, so as to avoid any issues related to spuriousness (see Threats to Validity below).

## 6.2 Evaluation of Effectiveness

Here we answer the first research question: "Is Metamorphic Runtime Checking more effective than using runtime assertion checking for detecting faults in applications without test oracles?"

Table 2 and Figure 3 summarize the results of the first experiment. 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, the number of equivalent mutants, and the different number of mutants that led to fatal errors or obviously wrong output. Overall, Metamorphic Runtime Checking was more effective, killing 1,602 (90.4%) of the mutants in the applications, compared to just 1,498 (84.5%) for assertion checking.

Broadly speaking, Metamorphic Runtime Checking was more effective at killing mutants that related to operations on arrays, sets, collections, etc. As an example, MartiRank

**Table 2: Results of mutation analysis for Metamorphic Runtime Checking (MRC) and runtime assertion checking (RAC).**

| | | Mutants Killed | |
|---|---|---|---|
| **Application** | **#Mutants** | **MRC** | **RAC** |
| C4.5 | 856 | 823 | 804 |
| GAFFitter | 66 | 36 | 20 |
| K-means | 35 | 28 | 26 |
| JSim | 36 | 36 | 36 |
| LDA | 24 | 24 | 24 |
| Lucene | 15 | 14 | 9 |
| MartiRank | 413 | 390 | 366 |
| PAYL | 40 | 29 | 13 |
| SVM | 287 | 222 | 200 |
| **Total** | **1,772** | **1,602** | **1,498** |

includes a function that evaluates the "quality" of a ranking by using a variant of the Area Under the Curve [19] metric for an array of items, and some mutations in the experiment caused the function to omit elements of the array. Daikon might detect that the quality is always a positive value, that the array does not change, etc., but none of these would be violated as a result of this mutation. However, the metamorphic property that permuting the elements in the array should not affect the output would be violated, if a different element were omitted upon additional invocation of the function, and in this case the bug would be found.

On the other hand, runtime assertion checking was more effective at killing mutants that affected the value of particular calculations. Such an example comes from the `FormTree` function used to create nodes of the decision tree in C4.5. Part of this function calculates the frequency with which a class (or "label") appears in the input data by looking at each item in the training data and updating a variable with the appropriate weight.

A mutation in this function such that the addition of the weight is changed to subtraction may cause the value of the variable to become negative, in violation of the invariant (detected by Daikon) that the frequency 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 training data values do not affect the calculation of class frequency, and even though the resulting output is incorrect, none of the metamorphic transformations cause the properties to be violated.

Further analysis could more specifically characterize the types of faults each approach is most suitable for detecting, but it follows, then, that runtime assertion checking and Metamorphic Runtime Checking should be used together in the testing of applications without test oracles. When used in combination in our experiments, they were able to kill 95% of the mutants: only 88 of the 1,772 survived.

## 6.3 Contributions to Effectiveness

We now address the second research question: "What contribution do application-level and function-level metamorphic properties make to the effectiveness of Metamorphic Runtime Checking?"

Table 3 shows the results with the mutants grouped by (a) those that were killed *only* by application-level metamorphic properties, (b) those that were killed *only* by function-level properties, (c) those that were killed by both types of properties, and (d) those that were not killed by either one.

Whereas application-level metamorphic properties killed a majority of the mutants (a combined total of 1465 mutants, or 82.6%, considering those only killed by application-level properties and those killed by both types), it is certainly a noteworthy result that function-level properties could kill an additional 137 mutants (7.7%) that went undetected by application-level properties. As described in the following paragraphs, this happened primarily because we were able to increase: the number of properties identified (scope); the number of tests run (scale); and the likelihood that a fault would be detected (sensitivity).

The improvement in the **scope** of metamorphic testing was particularly clear in the anomaly-based intrusion detection system PAYL. The Columbus framework was only to discover two application-level metamorphic properties because it was not possible to create new program inputs based on modifying the values of the bytes inside the payloads (say, increasing them), since the application itself only allowed for particular syntactically and semantically valid inputs that reflected what it considered to be "real" network traffic. These two properties were only able to kill two of the 40 mutants. However, once we could use Metamorphic Runtime Checking to run metamorphic tests at the function level, Columbus was able to identify many more properties that involved changing the byte arrays that were passed as function arguments, thus revealing 27 additional faults.

Likewise, we were able to increase the **scale** of metamorphic testing by running many more test cases. For instance, in MartiRank, even though Columbus discovered function-level properties for only a handful of functions, many of those are called numerous times per program execution, meaning that there are many opportunities for the property to be violated. A single execution of MartiRank consists of a number of "rounds" in which the set of input data is broken into sub-lists: there are $N$ sub-lists in the $N$th round, each containing $1/N$th of the total number of positive examples (i.e., examples with a label of 1). Each of the $N$ sub-lists is sorted by each attribute, ascending and descending, and then MartiRank determines the attribute and sorting direction for which the sorting gives the best "quality" ranking, as described above. In our tests, we ran MartiRank with the default 10 rounds, meaning that the functions to sort the elements and calculate the quality were each run over 100 times. Since many of the inputs had 100 attributes in the data set, and each function had four properties, that's over 40,000 test cases per function from just a single execution of the program!

Another reason why function-level properties were able to kill mutants not killed by application-level properties is that we were able to improve the **sensitivity** in terms of the ability to reveal more subtle faults, as seen in GAFFitter. In the function to calculate the "fitness" of a given candidate solution in the genetic algorithm, i.e., how close to the optimal solution (target) a candidate comes, one of the metamorphic properties is that changing the ordering of the elements in the candidate solution should not affect the result, since it is merely taking a sum of all the elements.

If, for instance, there is a mutation such that an last element is omitted from the calculation, then the metamorphic property will be violated since the return value will be

Table 3: Number of mutants killed by different types of metamorphic properties.

| Application | Total Mutants | Mutants Killed by: Application-level Properties Only | Function-level Properties Only | Both Types | Not Killed |
|---|---|---|---|---|---|
| C4.5 | 856 | 133 | 37 | 653 | 33 |
| GAFFitter | 66 | 2 | 14 | 20 | 30 |
| K-means | 35 | 6 | 11 | 11 | 7 |
| JSim | 36 | 14 | 0 | 22 | 0 |
| LDA | 24 | 2 | 0 | 22 | 0 |
| Lucene | 15 | 5 | 3 | 6 | 1 |
| MartiRank | 413 | 298 | 22 | 70 | 23 |
| PAYL | 40 | 0 | 27 | 2 | 11 |
| SVM | 287 | 69 | 23 | 130 | 65 |
| **Total** | **1,772** | **529** | **137** | **936** | **170** |

different after the second function call. However, at the application level, such a fault is unlikely to be detected, since the metamorphic property simply states that the quality of the solutions should be increasing with subsequent generations. Even though the *value* of the fitness is incorrect, it would still be increasing (unless the omitted element had a very large effect on the result, which is unlikely), and the property would not be violated.

One of the more interesting results of this experiment is that some of the newly discovered faults were in functions for which metamorphic properties had *not* been identified and thus were not being checked using function-level properties. These faults put the application into a state in which the metamorphic property of another function would be violated. For instance, the `pauc` function in MartiRank is passed an array of numbers and performs a calculation on them to determine the quality of the ranking, returning a normalized result between 0 and 1. One of the metamorphic properties of that calculation is that `pauc(A') = 1 - pauc(A)` where `A'` is the array in which the values of `A` are in reverse order. However, a fault in a *separate* function that deals with how the array was populated caused this property to be violated because the data structure holding the array itself was in an invalid state, even though the code to perform the calculation was in fact correct.

As a slight simplification, we can explain this as follows: the values in the array `A` were being stored in a doubly-linked list, so that MartiRank could calculate the "quality" of the list by looking at it forwards (ascending) and backwards (descending). A mutation in the function that created the linked list caused some of the links to "previous" nodes to point to the wrong ones. In this case, traversing the linked list in the forward direction would give `ABCDE`, but backwards would give `EDBCA`, for instance. The metamorphic property that `pauc(A) = 1 - pauc(A')` would only hold if `A'` were, in fact, the exact opposite ordering of `A`, but clearly in this case it is not, and Metamorphic Runtime Checking was able to detect this fault.

## 6.4 Performance Overhead

Last, we answer the research question, "Is Metamorphic Runtime Checking suitable for practical use?" by considering the performance impact of using such an approach.

Although Metamorphic Runtime Checking using function-level properties is able to detect faults not found by metamorphic testing based on application-level properties alone, this runtime checking of the properties comes at a cost, particularly if the tests are run frequently. In application-level metamorphic testing, the program needs to be run one more time with the transformed input, and then each metamorphic property is checked exactly once (just at the end of the program execution). In Metamorphic Runtime Checking, however, each property can be checked numerous times, depending on the number of times each function is called, and the overhead can grow to be much higher.

During the studies discussed above, we measured the performance overhead of our C and Java implementations of the Columbus framework. Tests were conducted on a server with a quad-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. On average, the performance overhead for the Java applications was around 3.5ms per test; for C, it was only 0.4ms per test. This cost is mostly attributed to the time it takes to create the sandbox and test process or thread.

This impact can certainly be substantial from a percentage overhead point of view if many tests are run in a short-lived program. For instance, for C4.5, the overhead was on the order of 10x, even though in absolute terms it was well under a second. However, we point out that, for most the programs we investigated in our study, the overhead was typically less than a few minutes, which we consider a small price to pay for being able to detect faults in programs without a practical, general test oracle.

## 6.5 Threats to Validity

In our study, we used Daikon to create the program invariants for runtime assertion checking. Although in practice invariants may typically be generated by hand, and some researchers have questioned the usefulness of Daikon-generated invariants compared to those generated by humans [38], we chose to use the tool so that we could eliminate any human bias or human error in creating the invariants. Additionally, others have independently shown that metamorphic properties are more effective at detecting defects than manually identified invariants [20], though for programs on a smaller scale than those in our experiment (a few hundred lines, as opposed to thousands as in many of the programs we used).

Daikon can generate spurious invariants if there are not

enough program executions or if there are bugs in the code (likewise, the same can be said about Columbus and the metamorphic properties it discovers), but we mitigated this by using the same inputs for the mutation analysis experiment as for the generation of invariants and properties. Regardless of whether these "should" hold for the various functions, any violation of them would necessarily be coming from the inserted mutation, since they were never violated by those inputs in the unmutated version.

Last, the ability of metamorphic testing to reveal failures is clearly dependent on the selection of metamorphic properties, and the results may have varied had we used different ones instead. However, we have shown that a basic set of metamorphic properties identified by Columbus can be used even without a particularly strong understanding of the implementation. Using this approach, therefore, we are demonstrating the *minimum* effectiveness of Metamorphic Runtime Checking; the use of domain-specific properties may actually reveal even more failures [50].

# 7. RELATED WORK

Although Metamorphic Runtime Checking is similar to runtime assertion checking, there are some key differences between metamorphic properties and program invariants. First, invariants and assertions address how variables relate to each other, or variable's expected values, but do not describe the expected relations between outputs that result from multiple executions with transformed inputs. Additionally, in practice assertions typically are read-only and do not have side effects; in our approach, though, the properties rely on additional invocations of the functions, which are allowed to have side effects, but these side effects are contained so as not to affect the running program.

Metamorphic properties are similar in some ways to algebraic specifications [13], though algebraic specifications often declare legal sequences of function calls that should produce an expected result, but do not describe how a particular function should react when its input is changed. The runtime checking of algebraic specifications has been explored in [35] and [42], though neither work considered the particular issues that arise from testing without oracles. Other work in this area has focused on consistency checking of abstract data types [43], but has not sought to create oracles for applications that do not otherwise have them.

Recent work in metamorphic testing has focused on its applicability in different domains (e.g., [40, 50]), but not on improving its general effectiveness in terms of scope, scale, and sensitivity. Beydeda [6] first brought up the notion of combining metamorphic testing and self-testing components, but did not investigate an implementation or consider the effectiveness on testing applications without oracles, as we do here.

Finally, the work presented here extends our own previous research into function-level metamorphic testing [32], by presenting a new execution model, a new technique for discovering likely properties, a new framework architecture, and the results of new empirical studies.

# 8. FUTURE WORK

Although the Columbus framework described above preserves the internal state of the application during function-level test execution, it does not prevent external side effects that may change the system, e.g., files, a database, etc. To address this, future work could consider integration with a record/replay system such as Chronicler [5] to record execution of the program up to the point at which the metamorphic testing would be conducted, and then execute the tests later and/or in a different environment so that potential changes to the system do not affect the original program execution.

It may also be possible to apply Metamorphic Runtime Checking to non-functional aspects of the program by specifying properties related to the system state – such as heap memory usage, file descriptors, etc. – or "security invariants" [14] that, if violated upon subsequent function invocations, indicate a vulnerability. In our mutation analysis experiments, we noticed that some mutations affected whether or not memory was freed, caused buffer overflows, etc. but did not change the program output and thus were considered "equivalent mutants". Metamorphic properties related to expected changes in system state may be able to detect such faults and vulnerabilities.

Finally, even though the applications studied in our experiment have no practical, general test oracle, some of their constituent functions certainly might, and conventional unit testing may be able to kill many of the mutations in the programs. Future research could investigate the effectiveness of combining traditional unit testing with Metamorphic Runtime Checking in such cases.

# 9. CONCLUSION

Metamorphic testing is an effective means of demonstrating faults in application domains without practical, general test oracles, such as machine learning, optimization, and scientific computing. In this paper, we have improved upon previous work in this area by introducing *Metamorphic Runtime Checking*, a new testing approach based on checking the metamorphic properties of both the entire application and its individual functions as the program runs. This paper has also described the architecture of a testing framework called *Columbus*, and discussed how it could be used to discover metamorphic properties.

As shown in our empirical studies, Metamorphic Runtime Checking is more effective than using runtime assertion checking, and has three distinct advantages over metamorphic testing using application-level properties alone. First, we are able to increase the scope of metamorphic testing, by identifying properties for individual functions in addition to those of the entire application. Second, we increase the scale of metamorphic testing by running more tests for a given input to the program. And third, we can increase the sensitivity of metamorphic testing by checking the properties of individual functions, making it possible to reveal subtle faults that may otherwise go unnoticed.

# 10. ACKNOWLEDGEMENTS

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

The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

## 11. REFERENCES


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

[2] Apache Lucene. http://lucene.apache.org.

[3] Apache Mahout. http://mahout.apache.org.

[4] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR01 -02, Dept. of Computer and Information Science, Univ. of Oregon, 2001.

[5] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proc. of the 35th International Conference on Software Engineering*, May 2013.

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

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, January 2003.

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

[9] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(24), 2009.

[10] T. Y. Chen, F.-C. Kuo, H. Liu, and S. Wang. Conformance testing of network simulators based on metamorphic testing technique. *Lecture Notes in Computer Science*, 5522, 2009.

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

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

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

[14] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing testing framework for software vulnerability detection. *International Journal of Secure Software Engineering*, 1(3):41–55, 2010.

[15] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, Sept. 2006.

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

[17] GAFFitter. http://gaffitter.sourceforge.net/.

[18] P. Gross, A. Boulanger, M. Arias, D. Waltz, P. M. Long, C. Lawson, R. Anderson, M. Koenig, M. Mastrocinque, W. Fairechio, J. A. Johnson, S. Lee, F. Doherty, and A. Kressner. Predicting electricity distribution feeder failures using machine learning susceptibility analysis. In *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, 2006.

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

[20] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse. An empirical comparison between direct and indirect test result checking approaches. In *Proc. of the 3rd International Workshop on Software Quality Assurance*, pages 6–13, 2006.

[21] U. Kanewala and J. M. Bieman. Techniques for testing scientific programs without an oracle. In *Proc. of the 2013 International Workshop on Software Engineering for Computational Science and Engineering*, 2013.

[22] U. Kanewala and J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. Technical Report CS-13-106, Computer Science Dept., Colorado State University, June 2013.

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

[24] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *Proc. of the 12th International Conference on Quality Software*, pages 59–68, August 2012.

[25] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[26] K. S. Misra and G. Kaiser. Effectiveness of teaching metamorphic testing. Technical Report cucs-020-12, Dept. of Computer Science, Columbia University, November 2012.

[27] MIT corpus of the complete works of William Shakespeare. http://shakespeare.mit.edu.

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

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

[30] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil. On effective testing of health care simulation software. In *Proc. of the 3rd International Workshop on Software Engineering in Health Care*, 2011.

[31] C. Murphy, K. Shen, and G. Kaiser. Automated system testing of programs without test oracles. In *Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA)*, pages 189–199, 2009.

[32] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proc. of the Second IEEE International Conference on Software*

*Testing, Verification and Validation (ICST)*, 2009.

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

[34] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242, 2002.

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

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

[37] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[38] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proc. of the 2009 International Symposium on Software Testing and Analysis (ISSTA)*, pages 93–104, 2009.

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

[40] A. Ramanathan, C. A. Steed, and L. L. Pullum. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *Proc. of the 2012 ASE/IEEE International Conference on BioMedical Computing*, pages 68–73, Dec. 2012.

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

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

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

[44] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 69–80, 2009.

[45] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: how i learned to stop worrying about fault propagation and love mutation testing. In *Proc. of the 34th International Conference on Software Engineering*, pages 870–880, May 2012.

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

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

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

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

[50] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Application of metamorphic testing to supervised classifiers. In *Proc. of the 9th International Conference on Quality Software (QSIC)*, pages 135–144, 2009.

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