Using Metamorphic Testing at Runtime to Detect Defects in Applications without Test Oracles

Thesis proposal

Christian Murphy
Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
+1.212.939.7184
cmurphy@cs.columbia.edu

Advisor: Gail E. Kaiser

December 3, 2008

Abstract

Assuring the quality of applications such as those in the fields of scientific calculations, simulations, optimizations, data mining, machine learning, *etc.* presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs".

Although knowing the correct output of such non-testable programs is impossible for arbitrary input, we have observed that even when there is no oracle in the general case, there can still be a limited subset of inputs for which the output can, in fact, be predicted. These are typically only very simple inputs, however, and may not have much power at revealing defects. Other inputs that, for instance, push boundary or timing limits can at least reveal gross errors, *e.g.*, catastrophic failures (crashes), but we require an approach that will reveal more subtle defects for arbitrary input.

Without an oracle, it is impossible to know what the expected output should be for any given input, but it may be possible to predict how changes to the input should cause changes to the output, and thus draw a relation between a set of inputs and the set of their respective outputs. We propose to build on an approach called "metamorphic testing", which takes advantage of properties of functions such that it is possible to predict expected changes to the output for particular changes to the input. If the change is not as expected, then a defect must exist.

In order to generate the test cases, metamorphic testing requires the initial input and output values, which could be generated using techniques like equivalence partitioning or random testing. However, inputs chosen using these techniques might miss some defects, since they might not happen to consider a sufficient variety of potential system states. Some defects in such systems may only be found under certain application states or for certain inputs that may not have been tested. Thus, we require a strategy that specifically considers these field states, by using real inputs and outputs from actual executions rather than just those generated in the testing lab.

Our approach, therefore, entails continuing metamorphic testing of the application as it runs in the deployment environment. For either an individual function or for the entire application, we first capture initial input/output pairs that are taken from actual executions in the field. Although we cannot know whether the output is correct (since there is no general test oracle), we at least know that the input is something that comes up in practice, and is thus useful as a valid test case. We then apply the function's or application's "metamorphic properties" to derive new test input, so that we should be able to predict the corresponding test output. Although we cannot know whether the test output is correct either, if it is not as predicted then there is a defect. Since this process is conducted in the field, we must ensure that users do not notice this testing, *e.g.*, see the test output, experience a sudden performance lag, *etc*.

The principal hypothesis is that, for programs that do not have a test oracle, conducting metamorphic testing within the context of the application running in the field can reveal defects that would not ordinarily otherwise be found. Additionally, we believe that this can be done without affecting the application state from the users' perspective, and with acceptable performance overhead. By way of proving these hypotheses, this thesis will make three contributions.

First, we will present an approach called *Automated Metamorphic System Testing*. This will involve automating 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 will allow for metamorphic testing to be conducted in the production environment without affecting the user, and will not require the tester to have access to the source code. The tests do

not require an oracle upon their creation; rather, the metamorphic properties act as built-in test oracles. We will also introduce an implementation framework called *Amsterdam*.

Second, we will present a new type of testing called *Metamorphic Runtime Checking*. This involves the execution of metamorphic tests from within the application, *i.e.*, the application launches its own tests, within its current context. The tests execute within the application's current state, and in particular check a function's metamorphic properties. We will also present a system called *Columbus* that supports the execution of the Metamorphic Runtime Checking from within the context of the running application. Like Amsterdam, it will conduct the tests with acceptable performance overhead, and will ensure that the execution of the tests does not affect the state of the original application process from the users' perspective; however, the implementation of Columbus will be more challenging in that it will require more sophisticated mechanisms for conducting the tests without pre-empting the rest of the application, and for comparing the results which may conceivably be in different processes or environments.

Third, we will describe a set of *metamorphic testing guidelines* that can be followed to assist in the formulation and specification of metamorphic properties that can be used with the above approaches. These will categorize the different types of properties exhibited by many applications in the domain of machine learning and data mining in particular (as a result of the types of applications we will investigate), but we will demonstrate that they are also generalizable to other domains as well. This set of guidelines will also correlate to the different types of defects that we expect the approaches will be able to find.

Contents

1 Introduction

Assuring the quality of applications such as those in the fields of scientific calculations, simulations, optimizations, data mining, *etc.* presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [102]. These applications 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"* [102].

Machine learning (ML) applications fall into this category as well. Formal proofs of an ML algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly. As these types of applications become more and more prevalent in society [66], ensuring their quality becomes more and more crucial. For instance, [51] lists over fifty different real-world applications, ranging from facial recognition to computational biology, that use the Support Vector Machines [96] classification algorithm alone. Additionally, ML ranking applications are widely used by Internet search engines [15], and intrusion detection systems are clearly becoming more and more important as important data is stored online and attackers seek to access it or gain control of systems.

Although knowing the correct output of such non-testable programs is impossible for arbitrary input, we have observed that even when there is no oracle in the general case, there can still be a limited subset of inputs for which the output can, in fact, be predicted. These are typically only very simple inputs, however, and may not have much power at revealing defects. Whereas other inputs, like those that push boundary or timing limits, can at least reveal gross errors, *e.g.*, catastrophic failures (crashes), we require an approach that will reveal more subtle defects for arbitrary input.

Without an oracle, it is impossible to know in the general case what the expected output should be for any given input, but it may be possible to predict how changes to the input should cause changes to the output, and thus draw a relation between a set of inputs and the set of their respective outputs. This approach is known as "metamorphic testing" [19], which is designed as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure. In metamorphic testing, if input x produces an output $f(x)$, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation function t, which can then be applied to the input to produce $t(x)$; this transformation then allows us to predict the expected output $f(t(x))$, based on the (already known) value of $f(x)$. If the output is not as expected, then a defect must exist. Of course, without an oracle this can only show the existence of defects and cannot demonstrate their absence, since the correct output cannot be known in advance (even if the outputs are as expected, both $f(x)$ and $f(t(x))$ could be incorrect), but metamorphic testing provides a powerful technique to reveal defects in such non-testable programs by use of a built-in "pseudo-oracle" [26].

In order to generate the test cases, metamorphic testing requires the initial input x and output $f(x)$ values, which could be generated using techniques like equivalence partitioning or random testing [30]. However, inputs chosen using these techniques might miss some defects, since they might not happen to consider a sufficient variety of potential system states. Some defects in such systems

may only be found under certain application states that may not have been tested: for large, complex software systems, it is typically impossible in terms of time and cost to reliably test all possible system states before releasing the product into the field. Another emerging issue is the fact that, as multi-processor and multi-core systems become more and more prevalent, multi-threaded applications that had only been tested on single-processor/core machines are more likely to start to reveal concurrency bugs [55]. Thus, we require a strategy that specifically considers these field states, by using real inputs and outputs from actual executions rather than just those generated in the testing lab.

Our approach, therefore, entails continuing metamorphic testing of the application as it runs in the deployment environment. For either an individual function or for the entire application, we first capture initial input/output pairs that are taken from actual executions in the field. Although we cannot know whether the output is correct (since there is no general test oracle), we at least know that the input is something that comes up in practice, and is thus useful as a valid test case. We then apply the function's or application's "metamorphic properties" to derive new test input, so that we should be able to predict the corresponding test output. Although we cannot know whether the test output is correct either, if it is not as predicted then there is a defect. Since this process is conducted in the field, we must ensure that users do not notice this testing, *e.g.*, see the test output, experience a sudden performance lag, *etc*.

This proposal will make three contributions:

First, we will present an approach called *Automated Metamorphic System Testing*. This will involve automating 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 will allow for metamorphic testing to be conducted in the production environment without affecting the user, and will not require the tester to have access to the source code. The tests do not require an oracle upon their creation; rather, the metamorphic properties act as built-in test oracles. We will also introduce an implementation framework called *Amsterdam*, which automates the process by which program input data is modified, multiple executions of the application with its different inputs are run in parallel, and the outputs of the executions are compared to check that the metamorphic properties are satisfied. This must be done in such a manner that the user only sees the results of the main (original) execution, and not from any of the others that are only for testing purposes.

Second, we will present a new type of testing called *Metamorphic Runtime Checking*. This involves the execution of metamorphic tests from within the application, *i.e.*, the application launches its own tests, within its current context. The tests execute within the application's current state, and in particular check a function's metamorphic properties. We will also present a system called *Columbus* that supports the execution of the Metamorphic Runtime Checking from within the context of the running application. Like Amsterdam, it will conduct the tests with acceptable performance overhead, and will ensure that the execution of the tests does not affect the state of the original application process from the users' perspective; however, the implementation of Columbus will be more challenging in that it will require more sophisticated mechanisms for conducting the tests without pre-empting the rest of the application, and for comparing the results which may conceivably be in different processes or environments.

Third, we will describe a set of *metamorphic testing guidelines* that can be followed to assist in the formulation and specification of metamorphic properties that can be used with the above approaches. These will categorize the different types of properties exhibited by many applications in the domain

of machine learning and data mining in particular (as a result of the types of applications we will investigate), but will also be generalizable to other domains as well. This set of guidelines will also correlate to the different types of defects that we expect the approaches will be able to find. In [71], we identified six categories of simple metamorphic properties: adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing part of the data; and, adding additional data. In this work, we will define more complex categories and then demonstrate that applications in other domains (perhaps optimization and simulation) exhibit such properties, too.

The rest of this proposal is organized as follows. Section 2 formalizes the problem statement, and identifies requirements that a solution must meet. In Section 3, we propose an approach to the solution, and specify our hypotheses. Sections 4 and 5 each look at different parts of the solution, including the model that we will use, a more detailed architecture, and the results of our initial feasibility studies. Related work is then discussed in Section 6. The proposal ends with a detailed research plan in Section 7, a list of expected contributions in Section 8, and finally the conclusion in Section 9. We also include two appendices to provide further detail of some previous work.

2 Problem, Definitions and Requirements

2.1 Definitions

This section formalizes some of the terms used throughout this proposal.

- An **error**, also referred to as a defect or bug, is the deviation of system external state from correct service state [48].
- A **fault** is the adjudged or hypothesized cause of an error [48].
- A **failure** is an event that occurs when the delivered functionality deviates from correct functionality. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function [48].
- The **development phase** includes all activities from presentation of the user's initial concept to the decision that the system has passed all acceptance tests and is ready to deliver service in its user's environment [48].
- The **development environment** refers to a setting (physical location, group of human developers, development tools, and production and test facilities) in which software is created and tested by software developers and is not made available to end users [48].
- A **deployment environment**, or use environment, refers to a setting in which software is no longer being modified by software developers and can conceivably be made available to end users. This environment consists of the physical location in which the software is used; groups of administrators and users; providers (humans or other systems) that deliver services to the system at its use interfaces; and the physical infrastructure that supports such activities [48].
- The **execution environment** of an application refers to the setting in which the software is running; it can either be the development environment or the deployment environment.

- A **test oracle** is an entity (either human or a system) that is used to determine whether a software component's output (including observable side effects) is correct, according to the specifications, for a given input (including the system state) [7].
- **Metamorphic testing** is a technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure, in order to try to find uncovered flaws. It is a methodology of reusing input test data to create additional test cases whose outputs can be predicted [19].
- The **metamorphic properties** of a function define the relationships by which an output can be predicted based on a transformation of the input [19].

2.2 Problem Statement

Despite advances in software testing, there still remains a certain class of applications that can be called "non-testable programs" [102] because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. Without an oracle, we cannot demonstrate correctness of the implementation, but we need a testing approach that can at least demonstrate the presence of defects. Such defects may occur at the unit level or at the system level.

We have observed that even when there is no oracle in the general case, there can still be a limited subset of inputs for which the output can, in fact, be predicted (by using a "niche oracle" [69]) and thus it is possible to know whether an output is right or wrong. Additionally, other inputs, such as those that push boundary or timing limits, can be used to reveal gross errors, *e.g.*, catastrophic failures (crashes). However, we require an approach that will reveal more subtle defects for arbitrary input, as opposed to obvious defects for a limited set of input.

This may be feasible if it is possible to know the expected relationships between sets of inputs and their corresponding outputs (instead of between a single input and its corresponding output). Such an approach may require an initial set of input data, which could be generated using various techniques. However, inputs chosen using these techniques might miss some defects, since they might not happen to consider a sufficient variety of potential inputs and subsequent system states. Some defects in such systems may only be found under certain application states that may not have been tested because of a lack of appropriate input. Thus, we require a strategy that specifically considers these field states, by using real inputs and outputs from actual executions rather than just those generated in the testing lab, in order to reveal defects in applications that do not have test oracles.

2.3 Requirements

A solution to this problem would need to address not only the issue of the absence of a test oracle, but also consider the multiple possible states under which an application may run. Such a solution should meet the following requirements.

1. **Reveal defects that would not otherwise be revealed**. The solution to this problem must be able to reveal defects that would not ordinarily or realistically be revealed with other current testing approaches, including static techniques. Although it may be impossible in the general case to indicate that a particular output is correct, the solution must at least be able to indicate whether the output is incorrect.

2. **Allow the tests to execute within various execution states**. Regardless of whether the tests address individual units or the entire system, tests should be conducted while the application is running so as to ensure that they pass in any situation. Tests that address units would need to be executed "from within", *i.e.*, the application determines when to conduct tests and launches the tests from its current context. System-level tests may treat the application as a black box but still consider different configurations and runtime environments.

3. **Support a variety of application types**. The approach should be able to support various types of applications, ranging from single-user standalone programs (*e.g.*, scientific calculations, desktop publishing, web browsing, *etc.*) to more complex multi-user applications (*e.g.*, a three-tier web server application), including those that do actually have test oracles. Although we present an approach that is general purpose, in this work we specifically limit our scope to machine learning applications, which can be categorized as either supervised (*e.g.*, classification and ranking) or unsupervised (*e.g.*, anomaly detection or data mining).

4. **Be configurable**. The software vendor or a system administrator should be able to configure the implementation of the approach (the testing framework) to control the frequency with which tests are to be run, when during the program execution the tests are to be run, how many tests can run concurrently, what to do if a test fails, *etc*.

5. **Allow for the easy creation/specification of tests**. The approach should allow software developers to easily create and specify the test cases, using familiar or easy-to-learn techniques.

6. **Report defects back to the software developers**. If a test fails and a defect is discovered, the framework must allow for feedback to be sent to the software developers so that the failure can be analyzed and, ideally, fixed. In addition to sending a notification of a discovered defect, the framework should also send back useful information about the system state so that it can be reproduced.

7. **Execute the tests without affecting the application's state from the users' perspective**. If the tests are to be executed in a running application with real users, this should be done without affecting the outcome of the program. The testing framework must ensure that any changes to the application's state or to the environment made by the tests are then undone, so that the application goes back to its original state (before the test was run) and can then proceed as normal. Another approach is to run the tests in a separate "sandbox", so that the tests can be run in parallel, and the test processes would not affect the original.

8. **Have low performance impact**. The user of a system that is conducting tests on itself during execution should not observe any noticable performance degradation. The tests must be unobtrusive to the end user, both in terms of functionality and any configuration or setup, in addition to performance.

3 Hypotheses and Proposed Approach

This section describes our proposed approach to solving the above problem. It also states the hypotheses that the thesis will investigate, and clarifies the scope of the work.

3.1 Proposed Approach

One popular technique for testing such non-testable programs is to use a "pseudo-oracle" [26], in which multiple implementations of an algorithm process the same input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [46].

However, even without multiple implementations, often these applications exhibit properties such that **if the input or system state were modified in a certain way, it should be possible to predict the new output, given the original output**. This approach is what is known as "metamorphic testing" [19].

For example, 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. When a new payload arrives, its byte distribution is then compared to that model, and anything deemed anomalous causes an alert [100]. For a particular input, it may not be possible to know *a priori* whether it should raise an alert, since that is entirely dependent on the model. However, if while the program is running we take the new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect must exist. This approach does not require an oracle for the particular input; it only requires the specification of the application's so-called "metamorphic properties".

As an example of the types of metamorphic properties that such applications may exhibit, in [71] we enumerated six different categories of properties, listed in Table 1. For instance, in many supervised ML applications that act on a set of training data, reversing or randomly permuting the order of the examples in the input should not affect the output (the "model"). In other cases, increasing the value of numeric fields by a constant should have a predictable effect on the output that is easy to calculate. Although this list is not yet comprehensive, it gives an indication of the types of metamorphic properties we intend to investigate.

Table 1: Classes of metamorphic properties

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

Next, we suggest that **continuing to execute tests in the field, after deployment, will provide sufficient test data and reveal defects that are dependent on the application state**. By executing tests from within the software while it is running under normal operations and use, additional defects that depend on the system state (or a combination of state and environment) will also be revealed. For testing individual units (functions), this approach requires the use of a new type of test that is

designed to be run from within the application, as it is executing. These are tests that ensure that the metamorphic properties hold true no matter what the application's state is, and regardless of its configuration or runtime environment. This would need to go beyond passive application monitoring (*e.g.*, [80]) and actively test the application as it runs in the field.

Our approach, therefore, entails continuing metamorphic testing of the application as it runs in the deployment environment. For either an individual function or for the entire application, we first capture initial input/output pairs that are taken from actual executions in the field. Although we cannot know whether the output is correct (since there is no general test oracle), we at least know that the input is something that comes up in practice, and is thus useful as a valid test case. We then apply the function's or application's "metamorphic properties" to derive new test input, so that we should be able to predict the corresponding test output. Although we cannot know whether the test output is correct either, if it is not as predicted then there is a defect. However, since this process is conducted in the field, we must ensure that users do not "notice" this testing, *e.g.*, see the test output, experience a sudden performance lag, *etc*.

3.2 Hypotheses

In our solution, metamorphic tests are executed in the context of the running application, as opposed to a controlled or blank-slate environment. Tests focused on individual units are run continuously as the application runs, at appropriate points in the program execution; system-level tests are conducted in parallel with the running application. Crucial to the approach is the notion that the test must not alter the state of the application from the users' perspective. In a live system in the deployment environment, it is clearly undesirable to have a test application altering the system in such a way that it affects the users of the system, causing them to see the results of the test code rather than of their own actions.

The main hypotheses investigated are as follows:

1. **For programs that do not have a test oracle, conducting metamorphic testing within the context of the application running in the field can reveal defects that would not ordinarily otherwise be found**. That is, the approach can reveal defects in non-testable programs that would not be found using metamorphic testing (or other techniques, for that matter) in the development environment, or through using other approaches that test software as it runs in the field.

2. **This can be done without affecting the application state from the users' perspective, and with acceptable performance overhead.** Users of the software would ideally not even know that any testing was being performed.

We will apply this approach to applications in the domain of machine learning (ML), focusing specifically on subtle computational defects that come about due to programming errors and misinterpretation of specifications, as opposed to gross defects (like system crashes or deadlocks) that are a result of untested deployment environments, configurations, *etc*. As machine learning applications become more and more prevalent in various aspects of everyday life, it is clear that their quality and reliability take on increasing importance. Thus, we hope to advance the state of the art in testing these types of applications, and indicate that our approach works for certain other types of non-testable programs as well.

4 Automated Metamorphic System Testing

This section describes our solution to addressing the system-level testing requirements. A discussion of our solution for unit testing can be found in Section 5.

4.1 Model

To support system-level testing, we will create a technique called *Automated Metamorphic System Testing*, which automates the process by which program input data is modified, multiple executions of the application with its different inputs are run in parallel, and the outputs of the executions are compared to check that the metamorphic properties are satisfied. This must be done in such a manner that the user only sees the results of the main (original) execution, and not from any of the others that are only for testing purposes.

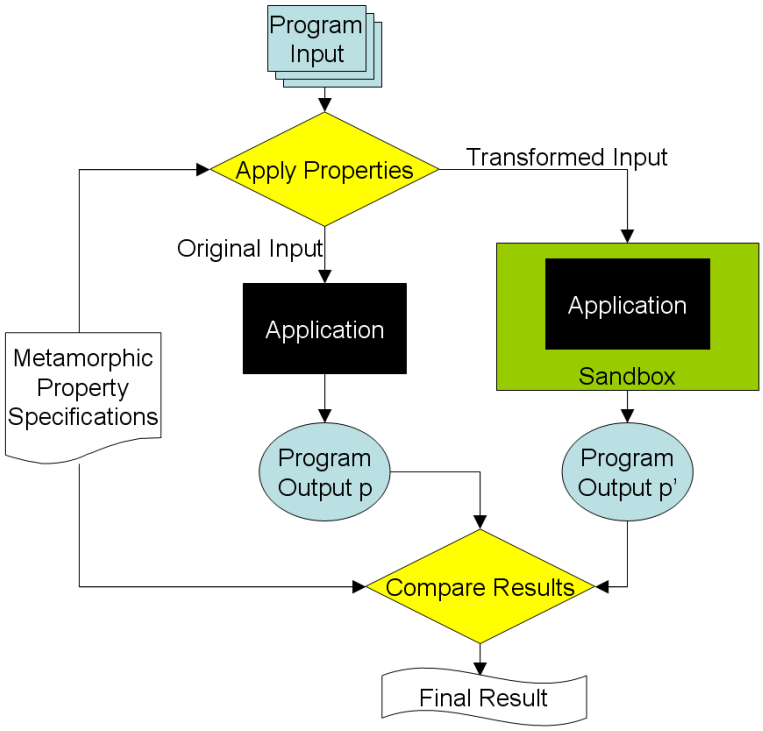

Figure 1: Model of Automated Metamorphic System Testing Framework

In the model of this approach, demonstrated in Figure 1, 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; depending on the metamorphic property, a modified version of this input data may also be produced. That 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.* are not seen by the user. When the two 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.

Note that the tester need not write any actual test code per se, but rather only needs to specify the metamorphic properties of the application. This can be done by the creator of the algorithm or by application designer, and does not assume intricate knowledge of the source code or other implementation details.

4.2 Architecture

This section describes the architecture of a framework, called *Amsterdam*, that will enable an application to be treated as a black box so that Automated Metamorphic System Testing can be performed without any modification to the code whatsoever. This framework allows the application to be tested as it runs in the field, using real input data. As described above, multiple invocations of the application are run and their outputs are compared; however, the additional invocations must not affect the user and must run in a separate sandbox.

4.2.1 Assumptions

The framework currently assumes that the program under test can be invoked from the command line, system input comes from files, and output is either written to a file or to standard out (the screen). Though this may limit the generality of this framework, according to our preliminary investigations, these assumptions are typically not restrictive in applications in the domain of interest. Additionally, when input comes from database tables, mouse clicks, keystrokes, incoming network traffic, *etc.*, the unit testing approach described in Section 5 can be used instead, since that inserts code into the application, and that code can perform any transformations of input and/or comparison of outputs at a more granular level.

4.2.2 Specifying Metamorphic Properties

The tester must first specify the metamorphic properties of the application. In our current prototype implementation of the framework, this will be done in an XML file; however, we are investigating other possibilities, such as using a formal specification language like Alloy [42], a scripting language like Python or Perl, or even plain-text English. The examples in this section assume the specifications are written in XML, 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 modify 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 be specified together.

For **input transformation**, the tester can describe how to modify (if modification is needed at all) the entire data set or only certain parts, such as a particular row or column in a table of data. In [71], we identified six 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; and, adding additional data.

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, and 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 for the input may need to be applied to the output before checking for equality. If the output is non-deterministic, the tester can specify a range of acceptable outputs, which may include some of the transformations, as well.

If the out-of-the-box transformation or comparison functionality of the framework is not expressive enough to meet the needs of a specific metamorphic property, the tester can add additional features by creating a separate component that can be invoked by the framework, according to a specific programming interface. For transformations, the input to this component would be the input data, and the output would be the modified data file; for comparisons, the input would be the two results to compare, and the output of the component would be whether or not the two results are as expected.

```
<TESTDESCRIPTOR>
  <EXECUTION>java NaiveBayes @parameters</EXECUTION>
  <PARAMETERS>-t @input.training_data -d @output.model</PARAMETERS>
  <INPUT>
    <VAR TYPE="arff_file" NAME="training_data" />
  </INPUT>
  <OUTPUT>
    <VAR TYPE="text_file" NAME="model" />
  </OUTPUT>
  <POST_TEST>
    <BRANCH OPTION="main" />
    <BRANCH OPTION="parallel" NAME="test1">
      @op_permute(@input.training_data)
    </BRANCH>
    <PROPERTY>
      <ASSERT> @op_equal(@main.output.model, @test1.output.model) </ASSERT>
    </PROPERTY>
  </POST_TEST>
</TESTDESCRIPTOR>
```

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

Figure 2 demonstrates an example of a metamorphic property for system testing as specified in an XML file. The input and output are given names, and the "post_test" specifies that there are to be two parallel executions and how to modify the inputs and compare the outputs. In particular, this file specifies that the NaiveBayes program (a ML classifier) has the property that, if the input ("training data") is perumuted, the output ("model") should still be the same. With minimal modification, the metamorphic properties specified in this XML file could also be applied to any other program that exhibits the same property.

4.2.3 Configuration

The tester then configures the framework to specify how the multiple invocations of the program should be executed. Although the framework is designed to be used in the production environment, it can certainly be used in the development environment as well. In these cases, parallel execution of the additional invocations and/or the use of a separate sandbox may not be required; thus, parallelism and sandboxing can be disabled, which may ease the process of debugging (if, for instance, the tester wants to see the traces of debugging statements printed to standard out). The tester may also want to specify whether or not the additional invocations should run on separate processors or cores, if supported by the underlying hardware.

The configuration also includes declaring what action to take if a metamorphic test fails. Because the test can only complete once all invocations of the program have completed, it would be too late to "interrupt" program execution, but the user can still be notified that the test revealed unexpected behavior, and results of the test can be sent back to the development team.

4.2.4 Execution of Tests

The testing framework first invokes the original application with the command line arguments specified in the property specification. While it is running, it then applies the specified transformations to the input files and creates temporary files to use for the additional invocations of the program. This is done after invoking the original application because modifying large files can take a long time, and there is no need for the original application to wait. The framework will provide 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). Other file formats can be supported by building custom transformation components.

The framework then starts additional invocations with the newly-generated inputs. The sandbox for the parallel processes is provided by a virtualization layer called a "pod" (PrOcess Domain) [82]. This creates a virtual environment in which the process has its own view of the file system and thus does not affect any other processes; additionally, the framework ensures that the user would not see any screen output created by one of the additional invocations of the program. However, at this time the framework sandbox does not include external entities such as the network or databases.

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.

4.2.5 Fault Localization

Localizing faults during system testing is quite challenging because the fault could come from anywhere in the code. Thus, this particular approach may be more suitable for detecting defects than for localizing them. However, we foresee an approach that combines coarse-grained Automated Metamorphic System Testing with fine-grained testing at the unit level, such that defects detected by the former can be localized with the latter. At this point, though, automated fault localization is outside the scope of work.

4.3 Feasibility

In [71] we performed system-level metamorphic testing of three machine learning applications: MartiRank [37], SVM-Light [43], and PAYL [99]. Although we did not automate this testing (*i.e.*, the modification of test input and the comparison of test output were either done manually or by using one-off scripts), we demonstrated that such an approach is feasible in revealing defects in applications in this domain.

Among our findings in that work, the most relevant regarding SVM-Light (an implementation of the Support Vector Machines [96] classification algorithm that could also be used for ranking) is that randomly permuting the order of the input data caused it to generate different results. The practical implication is that the order in which the data happens to be assembled can have an effect on the final outcome. The SVM algorithm theoretically should produce the same result regardless of the input data order; however, an ML researcher familiar with SVM-Light told us that because it is inefficient to run its quadratic optimization algorithm on the full data set all at once, the implementation performs "chunking" whereby the optimization algorithm runs on subsets of the data and then merges the results [94]. Numerical methods and heuristics are used to quickly converge toward the optimum; however, the optimum is not necessarily achieved, but instead this process stops after some threshold of improvement. Here the implementation deviates from the expected behavior.

We also tested PAYL, an anomaly-based network intrusion detection system, for which we did not have access to the source code. After analyzing PAYL's metamorphic properties, we conducted testing of PAYL by creating data sets (taken from actual network traffic), and then modifying them according to these metamorphic relationships. Our testing revealed two unrelated defects: in one, PAYL raised a different type of "alert" from what we expected; in another, it did not raise an expected alert at all. More importantly, as MartiRank and SVM-Light are examples of "supervised" machine learning, and PAYL is "unsupervised", this demonstrated that our testing approach would work on different types of machine learning applications.

All of these tests were performed using hand-crafted or random data sets, and we suspect that testing with data sets from actual usage could possibly reveal more defects. A testing framework will need to be created to automate this testing, as well as to make it plausible for use in the deployment environment so that tests can be run in parallel, but without the user noticing. This will be addressed in the thesis work.

5 Metamorphic Runtime Checking

This section describes the model and architecture of an approach that addresses metamorphic testing of individual units (functions) in a running application. Here we also discuss results of initial feasibility studies.

5.1 Model

In order to support metamorphic testing applied to individual functions, we introduce a technique called *Metamorphic Runtime Checking*. This entails a new type of tests that are to be executed in the running application, using the arguments to selected functions as they are called. The arguments are

modified according to the specification of the metamorphic properties, and the output of the function with the original input is compared to that of the function with the modified input; if the results are not as expected, then a defect has been exposed.

A simple example of a function to which Metamorphic Runtime Checking could be applied would be one 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 devation from the mean would still be the same (think about the values being "flipped" around the origin on the number line).

Furthermore, we know that there are other transformations that will alter the output, but in a predictable way. For instance, if each value in the set is multipled by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers, we can apply these "metamorphic properties" and create three more sets (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2), for a total of four test cases; moreover, given the result of only the first test case, we can predict what the other three should be.

Metamorphic testing generally would not be needed for this trivial example, but clearly can be very useful in the absence of an oracle: regardless of the values in the data set, and even if the correct output could not be known in advance, if the outputs are not as expected, then there must be a defect in the implementation. Although the use of these simple identities for testing numerical functions is not unique to metamorphic testing [25], the approach can be used on a broader domain of any functions that display metamorphic properties, including machine learning applications.

In our model, such tests are to be executed in the running application, using the arguments to these functions as they are called. For instance, in the standard deviation example, whenever the function is called, its argument can be passed along to a test method, which will multiply each element in the array by -1 and check that the two calculated output values are equal. This does not require a test oracle for the particular input; the metamorphic relationship specifies its own test oracle. It is true that if the two outputs are equal, they are not necessarily correct, but if they are not equal, then a defect must exist. This will allow us to not only execute tests in the field, within the context of the running application, but also to test those applications without a test oracle, by using the metamorphic tests themselves as built-in pseudo-oracles.

A framework that supports the execution of these tests has two primary requirements: run the tests from within the context of the running application; and do so without affecting that application's state from the users' perspective.

In our model of the testing framework, metamorphic tests are logically attached to the functions that they are designed to test. Prior to a function's execution, the framework invokes any corresponding test with some probability. In order not to have the user see the effects of the test, the testing framework will execute the metamorphic test in an isolated "sandbox", so that any changes to the state are not reflected in the original process. Additionally, the tests execute in parallel with the application: the test code does not preempt the execution of the application code, which can continue as normal. Figure 3 demonstrates the model we will use for conducting these tests.

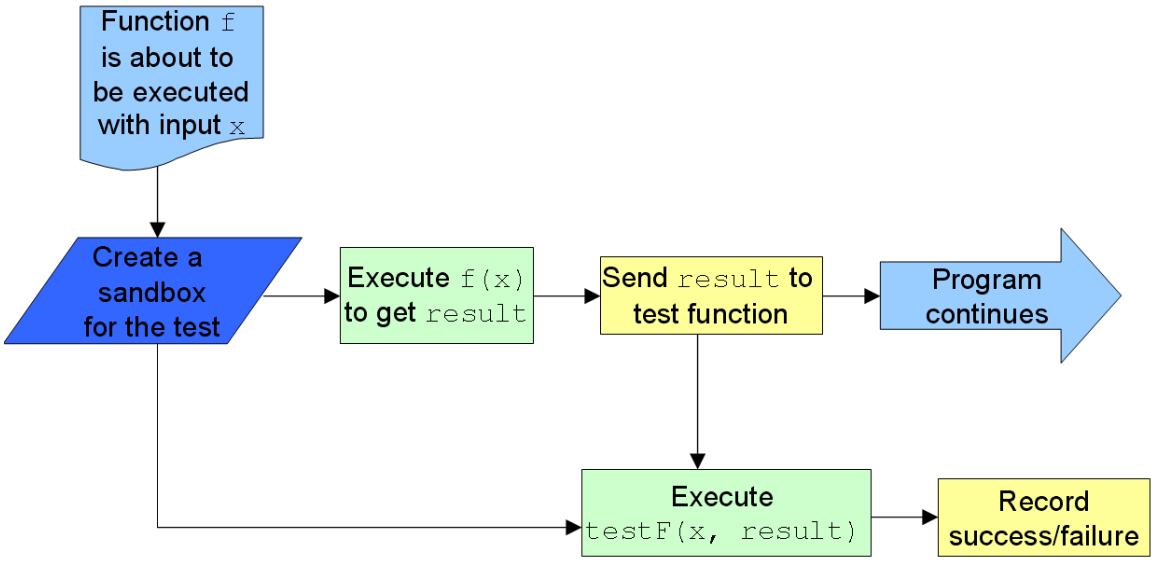

Figure 3: Model of Metamorphic Runtime Checking Framework

5.2 Architecture

In order to facilitate the execution of unit-level metamorphic tests in the deployment environment, we require a system that conducts the tests during actual runs of the application, using the same internal state as that of the original function. A system like Skoll [62] is a candidate for something on which to build, but it is primarily intended for execution of regression tests and determining whether builds and installs were successful, and not for testing the system as it runs; other assertion checking techniques (as surveyed in [24]) could be used, but they generally do not allow for calling the function again with different arguments (which we require), and typically pre-empt the execution of the rest of the program.

However, we have previously developed a testing technique called "In Vivo Testing" [70], in which tests are executed in the context of the running application without affecting the application state, specifically focusing on testing individual units (or combinations of units). For reasons of familiarity and simplicity, the Metamorphic Runtime Checking framework, called *Columbus*, will build upon the In Vivo Testing approach, which is described briefly below.

5.2.1 In Vivo Testing Overview

In Vivo Testing conducts tests "from within" the running application, using the current accumulated state of the component under test, as opposed to testing from a clean or constructed state, as is typical in unit testing [44]. Although existing unit and integration tests can be used with In Vivo Testing without any modifications (for instance, to address configurations or environments not tested prior to release, as in [62]), developers may find it desirable to create tests that ensure that properties of given subsystems or units hold true no matter what the application's state is. In the simplest case, they can be thought of as program invariants and assertions [24], though they go beyond checking the values of individual variables or how variables relate to each other, and focus more on the conditions that must hold after sequences of variable modifications and method calls, without worrying about side effects visible to the user.

A simple example is that of the functionality of an implementation of the Set interface in Java. One of its properties is that, if an object is added to the Set and then removed, a subsequent call to the "contains" method must return false. This condition must hold no matter what the state of the Set, and no matter what sort of object had been added. A traditional unit test may investigate this property by first creating a new, empty Set, but it would not be possible to conduct such a unit test on arbitrary states of the Set, after it has been used in a real, running application for some amount of time. Thus, an In Vivo test would be useful in this case.

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

As In Vivo tests are distinct methods run inside the application, the approach is like unit testing in the sense of calling individual methods with specified parameters, but it is also like integration testing in that it uses the integrated code of the whole application rather than stubs and drivers. In fact, In Vivo tests could be used in the development environment as well, and the creation of these tests may aid in the creation of further unit tests.

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

Further discussion of In Vivo Testing can be found in Appendix B. Note that In Vivo Testing is *not* part of the thesis work, but is only presented here for background purposes. Metamorphic Runtime Checking is similar in spirit to In Vivo Testing, in that tests execute within the context of the running application, and do not affect the state from the users' perspective, but other important details will be changed in order to faciliate the execution of metamorphic tests.

5.2.2 Assumptions

For In Vivo Testing, we currently assume that the application to be tested is written in an object-oriented language, running inside a managed execution environment, such as Java or C#. This assumption is made because the architecture is designed to test different components (classes) in isolation, and at a more granular level, to test individual methods prior to their execution. Additionally, the testing framework requires support for a "reflection" mechanism, so that the corresponding test functions can be called, given the name of the original function, and so that the function arguments can be

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

passed to the test function. However, we are currently implementing an architecture for Metamorphic Runtime Checking that will lift these two assumptions.

We also currently assume access to the source code, since the instrumentation of the functions is done at compile-time. Given that it is the software developers who will write the tests and instrument the code, we feel that this assumption is reasonable. However, as it may not always be possible or desirable to recompile the code, an approach to dynamically instrumenting the compiled code, such as in Kheiron [35] [36], could be used instead.

5.2.3 Creating Tests

The Columbus framework must be provided with test code that specifies the metamorphic properties to be checked within the running program. This test code would be written by the software developer (as opposed to a third-party developer or the end-user).

A test to be used with Metamorphic Runtime Checking is one that seeks to exercise the function's metamorphic properties, based on transformation of the input(s) and comparing the outputs. Figure 4 shows a metamorphic test for the Java implementation of a sine function, which exhibits two metamorphic properties: sin(α) = sin(α + 2π) and sin(α) = -sin(-α). The parameter "result" represents the return value of the original function call, so that outputs can be compared.

```
public boolean testSine(double angle, double result) {
  double s0 = this.sine(angle + 2 * Math.PI);
  double s1 = this.sine(-angle);
  return (s0 == result && s1 == -1 * result);
}
```

Figure 4: Example of Metamorphic Runtime Checking test

To aid in the generation of these tests, as explored in [72], we are investigating techniques to allow developers to specify metamorphic properties of a function using a special syntax in the comments. This way, developers do not have to write the tests, they only need to list the properties. Figure 5 shows such properties for an implementation of the sine function; these properties can then be used by a pre-processor in the testing framework to generate the test code shown above in Figure 4.

```
/**
 * @meta sine(angle + 2 * Math.PI) == \result
 * @meta sine(-angle) == -1 * \result
 */
public double sine(double angle) { ...
```

Figure 5: Specifying metamorphic properties

5.2.4 Instrumentation

After creating the tests, the software vendor must then select one or more functions in components (classes, files, *etc.*) of the application under test for instrumentation, such that instrumented function calls will be points at which a test could be run. Aside from acting as jumping off points for the tests, the instrumented functions are also the same ones that will be tested by the framework, and should be selected according to which ones the vendor wants to test (this could certainly be all of the functions,

of course). The list of functions is specified in an XML file. The selected functions are instrumented at compile-time, which does require access to the source code, thus this step is to be done by the software vendor. Note that this does not require any modification of the original source code; it only calls for recompilation.

5.2.5 Configuration

Before deployment, the administrator can configure the maximum number of concurrent tests that the system is allowed to execute at any given time. This prevents the testing framework from launching so many simultaneous tests that they flood the CPU and essentially block the main application. The administrator can also set a maximum allowable performance overhead, so that tests will be run only if the overhead of doing so does not exceed the threshold. The system tracks how much time it has spent running tests compared to how much time it has been running application code, and only allows for the execution of tests when the overhead is below the threshold.

Alternatively, the administrator can configure the framework so that, for each instrumented function with a corresponding test, there is a probability ρ with which that function's test will be run. This configuration is specified in an XML file, which contains the name of the component, the name of the function, and the percent of calls to that function that should result in execution of the corresponding tests. The file is read at run-time (not at compile-time) so it can be modified by a system administrator at the customer site if necessary. A "DEFAULT" percentage value can be specified as well: any function not explicitly given a percentage will use that global default. If the global default is not specified, then the default percentage is simply set to zero, which provides an easy way of disabling testing for all but the specified functions. To disable testing for all functions in the application, the administrator can simply set a "DISABLE" flag to true.

5.2.6 Execution of Tests

Before a function is to be called, the framework uses the measured performance overhead and/or the percentage value ρ for that function to decide whether to execute a test; it also checks whether the maximum allowed number of concurrent tests are currently executing. If a test is to be run and the function has a corresponding "test" function, this test will then be executed.

When a test is to be executed, a new process is first created as a copy of the original to create a sandbox in which to run the test code, ensuring that any modification to the local process state caused by the test will not affect the "real" application, since the test is being executed in a separate process with separate memory. This also ensures that the test function will get called at the same point in the program execution as the original function. The original function is called first, and then the function input and the result of the function call are passed as arguments to the test method in the sandbox process; within that function, the input can be modified and the outputs can be compared according to the metamorphic properties. Once the test function is invoked, the application can continue its normal execution, while the test runs in the other process. Note that the application and the test run in parallel in two processes: the test does not block normal operation of the application after the fork is performed. Depending on the configuration and the hardware, the test process may be assigned to a separate CPU or core.

To ensure that the metamorphic test does not make any changes to the file system, *etc.*, we have begun

to investigate integration with a thin OS virtualization layer that supports a "pod" (PrOcess Domain) [82] abstraction for creating a virtual execution environment that isolates the process running the test. However, the overhead of creating new "pods" may limit the effectiveness of the approach in the general case, so they will likely only be used for tests that actually affect the file system.

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

5.2.7 Handling Test Failure

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

5.2.8 Fault Localization

Because the approach tests individual functions, it will be clear which function's test failed, so fault localization could start there. However, it may not necessarily be the case that the function itself contains the defect. We have begun to investigate other fault localization techniques, as described in the Future Work section, though these are currently outside the scope of this particular work.

5.3 Feasibility

In [72], we investigated the feasibility of an approach in which we created an extension to the Java Modeling Language (JML) [49] to specify metamorphic properties; we then developed a tool called *Corduroy* to convert these properties into tests similar to those used in Metamorphic Runtime Checking, which were then executed using JML runtime assertion checking [12]. Details of the Corduroy implementation are described in Appendix A. Although the use of Corduroy is not part of the thesis work per se, this work demonstrates the feasibility of revealing defects by checking metamorphic properties at runtime, and much of the Columbus implementation will be based on features of Corduroy.

We applied the approach to some open-source Java applications that fall into the category of "non-testable programs". In particular, we looked at WEKA [103] and RapidMiner [86], which both provide Java implementations for numerous machine learning and data mining algorithms, and are popular tools for the development of Java machine learning applications.

Our testing involved the Naive Bayes, Support Vector Machines, K-Nearest Neighbors, and C4.5 implementations in WEKA 3.5.8, and the Naive Bayes implementation in RapidMiner 4.1. For each of these five applications, we first determined its metamorphic properties, using the approach described in [71]; note that this step did not even require viewing the source code or having knowledge of implementation details. We then annotated the corresponding methods with specifications using our extension to JML, used Corduroy to pre-process the source code, and then compiled it using the JML 5.6 compiler. Last, we used some of the data sets from the UC-Irvine Machine Learning Repository

[75] to perform our testing, using the JML 5.6 runtime environment to execute the code; no command line options were set for the machine learning applications, so all defaults were used. Our approach did not require the modification of any of the original application code, however some code needed to be added to facilitate our testing (see [72] for details).

We specified a total of 25 metamorphic properties for the five applications we investigated (see [72] for a complete listing), and our approach detected a defect in the calculation of confidence in RapidMiner's Naive Bayes classifier. The confidence value is a (normalized) indication of how sure the algorithm is about the classification it makes of examples in the classification phase. One would expect that if an example being classified had previously existed in the training data set and its confidence was c, and if the training data were modified so that the example existed twice, then upon classification the confidence should be $c/2$, since the algorithm would be twice as confident about its classification (a lower value means "more confident"). However, this turned out not to be the case. Further investigation revealed an error in one of the normalization calculations; this was a known defect in the version we tested, and was fixed in a later release.

Additionally, the KNN and Naive Bayes implementations in WEKA both provide an API for updating a model after it has been created by adding a new instance to the training data: we would expect that if training data set T produces model M, and if there is an example e such that training data set $T' = T - e$, and T' produces model M', then when M' is updated using e, it becomes equal to M. We discovered that the KNN implementation exhibits this property, but in WEKA's Naive Bayes implementation, the model created from a data set after it is updated with one example is sometimes (but not always) different from a model created from a data set containing that original example. Moreover, we observed that if a data set is updated with multiple examples, the number of differences between the updated model and a model created from a data set already including those examples had no correlation to the number of updates. When we inspected the code, we discovered that the update method does not correctly update the probability estimates, thus causing a difference compared to the model built using the entire data set.

We did notice an inconsistency in the WEKA SVM implementation (similar to the one described above for SVM-Light), but our testing did not demonstrate any defects in the KNN or C4.5 implementations in WEKA; although the tests cannot demonstrate correctness, either, since the correct output cannot be known in advance, the fact that the tests passed at least increases confidence in the implementations.

Note that the work here did not specifically use the In Vivo Testing framework or the proposed Columbus framework, but rather used the JML runtime environment to execute the tests. Using the JML runtime environment will not suit our needs going forward since, obviously, it only supports Java applications; additionally, the runtime checking in JML is not done in parallel, but is rather done sequentially, *i.e.*, checking the properties pre-empts the rest of the program. Additionally, JML does not generally allow the runtime checks to call functions with side effects, thereby limiting the types of functions we can test. However, this work demonstrates the feasibility of the Metamorphic Runtime Checking approach, as the tests were all executed from within the context of the running application, taking advantage of the functions' metamorphic properties.

6 Related Work

6.1 Addressing the Absence of a Test Oracle

Baresi and Young's 2001 tech report [7] presents a comprehensive overview of different approaches to addressing "the oracle problem", *i.e.*, the lack of a reliable test oracle in the general case. The different techniques are summarized here:

- Embedded Assertion Languages. Programming languages such as ANNA [56] and Eiffel [64], as well as C and Java, have built-in support for assertions that allow programmers to check for properties at certain control points in the program. In Metamorphic Runtime Checking, the tests can be considered runtime assertions; however, approaches using assertions typically address how variable values relate to each other, rather than considering how a function should react when its inputs are changed. Additionally, the assertions in those languages are not allowed to have side effects; in our approach, the tests are allowed to have side effects (in fact they almost certainly will, since we're calling the function again), but these side effects will be hidden from the user. Last, assertions typically pre-empt the rest of the application by running sequentially with the rest of the program [24], whereas in Metamorphic Runtime Checking the properties are checked in parallel.

- Extrinsic Interface Contracts. These are similar to assertions except that they keep the specifications separate from the implementation, rather than embedded within. Examples include languages like ADL [93] or techniques such as specifying algebraic specifications [25]. However, algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (*e.g., pop(push(X)) == X* in a Stack) [92], and are not as powerful for system-level testing. The runtime checking of algebraic specifications has been explored in [77] and [91], though neither work considered the particular issues that arise from testing without oracles. Others have looked at the automatic detection of algebraic specifications, in particular [40], and of program invariants in general (*e.g.*, DIDUCE [39], Daikon [32], Houdini [33], *etc.*); the automatic detection of metamorphic properties is outside the scope of this work.

- Pure Specification Languages. Formal languages like Z [1] or Alloy [42] can be used to declare the specific properties of the application, typically in advance of the implementation to communicate intended behavior to the developers. However, Baresi and Young point out that a challenge of using specification languges as oracles is that "effective procedures for evaluating the predicates or carrying out the computations they describe are not generally a concern in the design of these languages", *i.e.*, the language may not be powerful enough to describe how to know whether the implementation is meeting the specification. Additionally, as pointed out in [92], the specifications need to be complete in order to be of practical use in the general case.

- Trace Checking and Log File Analysis. Observing the execution of an application may indicate whether or not it is functioning correctly, if for instance it is conforming to certain properties (like a sequence of execution calls or a change in variable values) that are believed to be related to correct behavior; or, conversely, to see if it is *not* conforming to these properties. We have, in fact, investigated this technique previously with some success [69], but noted that in some cases 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.

While our approaches address some of the limitations described above, we recognize that metamorphic testing cannot provide a complete oracle, since there may still be some types of defects that cannot be detected using this technique. However, the use of metamorphic testing can complement these approaches and improve upon them, especially if specifying the metamorphic properties can be done easily.

6.1.1 Metamorphic Testing

Applying metamorphic testing to situations in which there is no test oracle has previously been studied by Chen *et al.* [21]. In some cases, these works have looked at situations in which there cannot be an oracle for a particular application [22], as in the case of "non-testable programs"; in others, the work has considered the case in which the oracle is simply absent or difficult to implement [17]. However, this previous work has mostly looked at system-level testing as opposed to internal unit- and integration-level testing as we present here. Additionally, whereas their work has primarily focused on functions with simple numerical input domains [20], we are working with inputs that conceivably consist of larger, alphanumeric data sets, as a result of the types of applications we are investigating.

Beydeda [10] first brought up the notion of combining metamorphic testing and self-testing components so that an application can be tested at runtime, but did not investigate an implementation or produce any results. [2] Our work extends that initial idea by providing implementation details and evidence of feasibility.

Gotleib and Botella [34] coined the term "automated metamorphic testing" to describe how the process can be conducted as the program runs, but they do not describe any mechanism for addressing performance concerns or for ensuring that the additional invocation of the function or the program is not seen by the user (*i.e.*, their approach was targeted at the development environment, whereas we target the deployment environment). Additionally, they only provided means for automating the testing of programs written in C; our Automated Metamorphic System Testing framework can be used with programs written in any language, and the Metamorphic Runtime Checking can be used for both C and Java programs.

6.1.2 Testing ML Applications

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (*e.g.*, [14], [18], [106], *etc.*), we are not currently aware of any work in the reverse sense: applying software testing techniques to machine learning applications, particularly those that have no reliable test oracle. Orange [27] and WEKA [103] are two of several frameworks that aid ML developers, but the testing functionality they provide is focused on comparing the quality of the results, and not evaluating the "correctness" of the implementations. Repositories of "reusable" data sets have been collected (*e.g.*, the UCI Machine Learning Repository [75]) for the purpose of comparing result quality, *i.e.*, how accurately the algorithms predict, but not for the software engineering sense of testing.

[2] In fact, I contacted Dr. Beydeda and he said that this paper was the only work he performed in this area, and he is not researching it any further.

Testing of intrusion detection systems [61] [76], intrusion tolerant systems [57], and other security systems [5] 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 IDS 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.

6.2 Runtime Testing

6.2.1 Perpetual Testing Approaches

Our investigation into the runtime checking of metamorphic properties is inspired by the notion of "perpetual testing" [83] [87] [88] [105], which suggests that analysis and testing of software should not only be a core part of the development phase, but also continue into the deployment phase and throughout the entire lifetime of the application. Perpetual testing advocates that analysis and testing should be on-going activities that improve quality through several generations of the product, in the development environment (the lab, or "in vitro") as well as the deployment environment (the field, or "in vivo"). Both testing approaches we suggest here (Automated Metamorphic System Testing and Metamorphic Runtime Checking) are types of perpetual testing in which the tests are executed in the field and do not alter the state of the application from the user's perspective.

Our testing is also a form of "residual testing" [84]. This type of testing is motivated by the fact that software products are typically released with less than 100% coverage, so testers assume that any potential defects in the untested code (the residue) occur so rarely so as not to bear consideration. Much of the research in this area to date has focused on measuring the coverage provided by this approach by looking at untested residue [74] [84] or by comparing the coverage to specifications [73]. However, this work does not consider the actual execution of tests in the deployment environment, as we describe here. Those approaches describe measurements of the residue, whereas we are attempting to discover the residual bugs by conducting tests. Our approach does not currently address coverage, but could be extended to do so, *e.g.*, emphasizing testing of the residue but not restricting the testing to only the residue, since bugs could reside in already-tested code.

Also related to perpetual testing is "continuous testing", which refers to round-the-clock execution of tests, though typically in the development environment [89] [90]. However, the Skoll project [47] [62] has extended this into the deployment environment by carefully managed facilitation of the execution of tests at distributed installation sites, and then gathering the results back at a central server. The principal idea is that there are simply too many possible configurations and options to test in the development environment, so tests can be run on-site to ensure proper quality assurance. Whereas the Skoll work to date has mostly focused on acceptance testing of compilation and installation on different target platforms, our testing is different in that it seeks to execute tests within the application while it is running under normal operation. Rather than check to see whether the installation and build procedure completed successfully, as in Skoll, we seek to execute tests as the application runs in its deployment environment.

6.2.2 Monitoring

Other approaches to testing software in the field include the monitoring, analysis, and profiling of deployed software, as surveyed in [31]. One of these, the GAMMA system [79] [80], uses software

tomography for dividing monitoring tasks and reassembling gathered information; this information can then be used for onsite modification of the code (for instance, by distributing a patch) to fix defects. Liblit's work on Cooperative Bug Isolation [50] (CBI) enables large numbers of software instances in the field to perform analysis on themselves with low performance impact, and then report their findings to a central server, where statistical debugging is then used to help developers isolate and fix bugs. Note that neither of these actually conducts active tests in the field as the In Vivo Testing approach does (and as our two proposed approaches will), but rather these other approaches deal with coverage monitoring (GAMMA) or observing variable values or the values returned by function calls (CBI). This obviates the need for any type of test oracle, but means that neither approach can be certain that a defect has been found (except for situations like crashes). On the other hand, the approaches described in this proposal do not need oracles either, but can reveal more subtle defects.

6.2.3 Self-Checking Software

While the notion of "self-checking software" is by no means new [104], much of the recent work in executing tests in the field has focused on COTS component-based software. This stems from the fact that users of these components often do not have the components' source code and cannot be certain about their quality. Approaches to solving this problem include using retrospectors [52] to record testing and execution history and make the information available to a software tester, and "just-in-time testing" [53] to check component compatibility with client software. Work in "built-in-testing" [100] has included investigation of how to make components testable [9] [11] [13] [60], and frameworks for executing the tests [28] [59] [63], including those in embedded systems [85] and Java programs [29], or through the use of aspect-oriented programming [58].

In light of all these important contributions, our testing approaches differentiate themselves by providing the ability to test any arbitrary part of the system (not just COTS components) and allowing for the easy creation of new tests, rather than requiring extensive modification to the original source to provide special functional and testing interfaces [3] [97] or enforcing a rearchitecture of the application to allow for the use of testers and controllers/handlers [6] [67] [97]. The advantage of our testing approaches over these others is that we are providing a framework for perpetual testing of an existing application with minimal modification, as opposed to prescribing a methodology for developing an application so that it may be tested after its deployment.

7 Research Plan and Schedule

7.1 Development Tasks

To support Automated Metamorphic System Testing, we will develop a framework called Amsterdam that supports the modification of program input, parallel sandboxed execution, and automatic comparison of the outputs, based on easy-to-write specifications of metamorphic properties. This framework will be implemented in Java, but does not require that the program being tested also be written in Java: that program is just invoked from the framework.

We will also develop an implementation of the Columbus framework for Metamorphic Runtime Checking for both Java programs and for C programs. Because of certain technical limitations of the existing In Vivo Testing framework, Columbus will require a new architecture: specifically, Columbus

needs to compare the outputs of the multiple function executions, which is not currently supported in In Vivo Testing.

Additionally, further enhancements will need to be performed, particularly relating to performance overhead and making the frameworks more efficient. To reduce the performance impact, the Columbus framework will no longer use the aspect-oriented programming language AspectJ [2], which is used in the In Vivo Testing framework, since it provides additional overhead for features that are not used, and our initial investigations into using AspectC for C programs showed even worse performance impact. Also, we will no longer use Java Reflection in determining the names of the tests to execute, since this also adds additional overhead. Rather, the Columbus framework will depend on source code pre-processing so that the creation of the test process and the calling of the test function can be more efficient.

For simplicity, the initial Columbus prototype will use a simple process forking mechanism for creating the parallel "sandbox" in which to run tests. We will then investigate the integration with the PODs [82] technology for virtualizing the test process, though from our initial investigations we know that the use of PODs will incur a greater performance penalty, so we will need to determine the tradeoffs of using such an approach. Specifically, in some cases the use of PODs will not be necessary (like if the test only affects the in-process memory), so we may need some mechanism of indicating when to use PODs and when a simple fork is sufficient.

The Amsterdam and Columbus frameworks are designed to be independent of each other, but could conceivably be used in conjunction, as there may be benefit in enabling tighter integration, such as for fault localization.

7.2 Experiments and Methodology

After the various implementations are completed and improved, we will conduct experiments to prove the hypotheses described above.

7.2.1 Demonstrating Effectiveness of the Approaches

To address the first part of the hypothesis and demonstrate that our approaches can reveal defects in applications for which there is no test oracle, we will select real-world machine learning applications, identify their metamorphic properties, and instrument them with the frameworks. We will then conduct both Metamorphic Runtime Checking and Automated Metamorphic System Testing on these applications as they run under normal operation in the field, and we expect that we will reveal new defects that were not previously known; we will also show that these defects would not ordinarily have been detected by using metamorphic testing prior to deployment.

For these experiments, we have identified numerous candidates that are used in real-world applications. One possible application is MEDUSA [65], a tool for computational biology that is included as a component of the geWorkbench [16] framework. We may also use the intrusion detection systems ANAGRAM [98] and Spectrogram [95], which could be used to monitor real traffic on the Columbia Computer Science department network. We may choose other applications besides these, but it is important that the applications come from different problem domains and/or use different types of machine learning algorithms, *i.e.*, supervised vs. unsupervised.

We will conduct tests on the "as-is" implementations to see if it is possible to detect previously-unknown defects, but also use techniques such as reproducing previously-discovered defects or, as a last resort, inserting defects (for instance via operator mutation) to further demonstrate feasibility as necessary.

7.2.2 Demonstrating Advancement of State-of-the-Art

To show that our testing approach advances the state of the art in testing applications that have no test oracle, we will compare it to other techniques that seek to address this same problem. In particular, we will show that symbolic execution may detect that only some simple metamorphic properties are satisfied (especially those that are based on simple modification of numeric values) but cannot determine whether the implementation meets other properties, such as those depending on state or those that involve the manipulation of data structures. We will also show that the use of runtime checking of program invariants alone is not enough to reveal defects in these types of applications, since a function may be satisfying its pre- and post-conditions, but still not be producing the correct output.

Additionally, we will investigate whether formal specification languages could be used as pseudo-oracles for these applications (assuming they are complete, which may be an undecidable problem [92]), and show that the specification of metamorphic properties is simpler, if not equally as effective. These will be "pencil-and-paper" exercises instead of empirical studies, but will demonstrate that in some cases there are certain defects that our approach can reveal that cannot be shown using these other techniques.

7.2.3 Evaluating Success Criteria

Aside from demonstrating effectiveness (ability to reveal defects) and showing that the approaches can reveal certain types of defects that other approaches cannot, we will attempt to determine the *adequacy* [101] of our testing approach. Typical metrics used in software testing such as statement coverage, defects per kloc per unit time, *etc.* [45] may not be as meaningful here since they depend heavily on the quality of the individual *test sets* (in our case, the specification of the metamorphic properties) and on the particular input that is used, and not on the quality of the testing *approach* itself. We note, though, that such metrics *can* be used to determine stopping points for the conducting of metamorphic testing, for instance when "enough" statements have been covered or when "enough" defects have been found. Additionally, there is nothing about our approach that prevents the use of these metrics for measuring the adequacy of individual test sets, though that is not our goal here.

Likewise, for a given test set, we cannot compare the adequacy of our approach against others like random testing [30], which requires a test oracle that does not exist in these cases. Also, existing testing benchmarking suites (like the Siemens test suite [107]) do not include applications without test oracles, and as described above rely on the individual test cases for determining adequacy. Thus, we will need to develop a new metric that allows us to compare our approach to others (such as the ones discussed above) that seek to test similar applications, given a particular test case (or set of test cases). For instance, such a metric could compare the false positive/negative rates of our approach and, say, symbolic execution in trying to determine whether a given function exhibits a particular metamorphic property.

7.2.4 Demonstrating Acceptable Performance Impact

Proving the second part of the hypothesis includes measuring the performance overhead of our approaches, to demonstrate that testing can be conducted with acceptable impact on the user. For the Automated Metamorphic System Testing, we will conduct these measurements during our tests demonstrating effectiveness, as described above.

For Metamorphic Runtime Checking, which has configurable options that affect the performance impact, we will first need to determine what "acceptable" actually means for the different types of applications. We will limit these applications to machine learning software as described previously. We will then benchmark the Metamorphic Runtime Checking framework by conducting experiments that will demonstrate the performance overhead of different scenarios. In particular, we will measure the actual incurred overhead when the user specifies an acceptable overhead level (to see how close we can reliably get to the expected value), and also the actual overhead when the user specifies different percentage values for the probabilities of running tests for individual functions.

Last, we will define metrics to measure the efficiency of the Metamorphic Runtime Checking approach, such as the number of conducted tests divided by the percentage overhead. The thought here is that, over a given period of time, conducting 100 tests with 10% overhead is equally as "good" as running 200 tests with 20% overhead, and is twice as "good" as running 50 tests with 10% overhead or 100 tests with 20% overhead. Other metrics could incorporate resource utilization, for example CPU usage, memory usage, disk space usage, *etc*. Note that these metrics do not deal with the efficiency of the discovery of defects (which are addressed above), but only focus on the execution of tests and the performance impact from the users' perspective.

7.2.5 Categorizing Defects and Demonstrating Suitability to Other Domains

Finally, part of our work will be to identify the different types of defects for which the approach is most suitable, and then to categorize the types of metamorphic properties that applications in the domain of machine learning may have. We will use some of the traditional software testing metrics described above to demonstrate that these categories of properties are adequate for testing the applications in the domain of interest.

We will then generalize these categories of metamorphic properties to other problem domains, and demonstrate that other applications (possibly in optimization, simulation, or scientific computing) have similar properties and that such testing approaches can be used to reveal similar types of defects in those domains, too.

7.3 Schedule

Table 2 shows my plan for completion of the research.

8 Expected Contributions

The contributions of this thesis are anticipated to include:

1. An approach called **Automated Metamorphic System Testing**. This will involve automating system-level metamorphic testing by treating the application as a black box and checking

Completion Date	Work	Progress
Aug. 2008	Finish initial Corduroy prototype (Java)	completed
Sep. 2008	Conduct Corduroy feasibility studies	completed
Oct. 2008	Submit Corduroy paper to ICST 2009	submitted
Nov. 2008	Write Thesis Proposal	completed
Dec. 2008	Defend Thesis Proposal	scheduled
Dec. 2008	Finish Columbus prototypes (Java, C)	ongoing
Dec. 2008	Complete initial Amsterdam prototype	ongoing
Jan. 2009	Conduct Columbus feasibility studies	ongoing
Jan. 2009	Conduct Amsterdam feasibility studies	ongoing
Jan. 2009	Submit Amsterdam paper to ISSTA 2009	
Jan. 2009	Integration with PODs	
Mar. 2009	Testing in live environments, including performance testing	
May. 2009	Compare approaches to use of formal specification languages and other static techniques	
Jun. 2009	Tests to determine adequacy and effectiveness	
Jul. 2009	Determine guidelines for creating metamorphic properties and identify types of defects for which the approaches are most suitable	
Aug. 2009	Demonstrate applicability to other domains	
Aug. 2009	Start writing Thesis	
Sep. 2009	Submit paper to ICSE 2010	
Oct. 2009	Submit additional paper to ICST 2010	
Jan. 2010	Submit additional paper to ISSTA 2010	
Apr. 2010	Defend Thesis	

Table 2: Plan for completion of research

that the metamorphic properties of the entire application hold after execution. This will allow for metamorphic testing to be conducted in the production environment without affecting the user, but will not require the tester to have access to the source code. We will also present an implementation framework called Amsterdam.

2. A new type of testing called **Metamorphic Runtime Checking**. This involves the execution of metamorphic tests from within the application, *i.e.*, the application launches its own tests, within its current context. The tests execute within the application's current state, but in particular check a function's metamorphic properties as well. These do not require an oracle upon their creation; rather, the metamorphic properties act as built-in test oracles. We will also present a system called Columbus that supports the execution of the Metamorphic Runtime Checking from within the context of the running application. Like Amsterdam, it will conduct the tests with acceptable performance overhead, and will ensure that the execution of the tests does not affect the state of the original application process.

3. A set of **metamorphic testing guidelines** that can be followed to assist in the formulation and specification of metamorphic properties that can be used with the above approaches. These will

categorize the different types of properties exhibited by many applications in the domain of machine learning in particular (as a result of the types of applications we will investigate), but will also be generalizable to other domains as well. This set of guidelines will also correlate to the different types of defects that we expect the approaches will be able to find.

In addition to the contributions listed above, at the time of this writing the following practical accomplishments have already been made:

- Published and presented a short version of this proposal at the FSE 2008 Doctoral Symposium [68].
- Published and presented a paper on system-level metamorphic testing of ML applications and properties for use in metamorphic testing at SEKE 2008 [71].
- Published and presented a paper on distributed In Vivo Testing at ICST 2008 [23].
- Presented a poster on In Vivo Testing at ISSTA 2008.
- Submitted a paper on In Vivo Testing to ICST 2009 [70].
- Submitted a paper on runtime metamorphic testing to ICST 2009 [72].
- Submitted a journal paper that included some results of our metamorphic system testing approach [69].

9 Future Work and Conclusion

There are a number of interesting future work possibilities, both in the short term and further into the future.

9.1 Immediate Future Work Possibilities

- **Ensure that the test code does not modify anything external to the system.** Currently the "sandbox" in which the test code runs only includes in-process memory and (assuming we use PODs) the file system, but it does not account for relational databases, network packets, legacy systems, *etc*. Tests that rely on these external entities would need to either "clean up" after themselves or be run in a special application-supported transactional mode.
- **Addressing non-determinism.** Some ML algorithms are intentionally non-deterministic and necessarily rely on randomization, which makes testing very difficult; our testing to date has been assisted by the fact that it is possible to "turn off" any randomization options in the applications we investigated. Although the frameworks will support the specification of properties to address limited types of non-determinism (such as output values falling within a range or in a set), more detailed analysis may be needed for future work on machine learning algorithms that depend on randomization.
- **Support automatic fault localization.** Although the system 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 like the ones described in [50] or [4] could be used to try to isolate the fault.

9.2 Possibilities for Long-Term Future Directions

- **Automatically detect properties that can be used to aid in the generation of tests.** Although prior work has been done in automatically determining algebraic specifications, *e.g.*, [40], and in categorizing metamorphic properties [71], as of now it is necessary for the software developer to discover and specify the properties and/or write the tests required for metamorphic testing. It may be possible to automate this process.
- **Explore soundness of metamorphic properties.** Others have demonstrated that, at the risk of false positives, when using model-based testing approaches, an unsound model (or, in our case, unsound metamorphic properties) may reveal defects that more restrictive sound properties would not [38]. For instance, in [69], we pointed out a metamorphic property in the ML ranking algorithm MartiRank [37] 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 (because 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), the new output will in general be *approximately* equal to the original, based on some metric of comparing rankings, such as the number of elements ranked differently, the Manhattan distance, or the Euclidean distance in N-dimensional space. At the expense of revealing false positives, this property may also reveal actual defects that may not be detected if we included the original constraint that all values must be distinct.
- **Enable collaborative defect detection and notification.** Aside from just sharing the performance load of conducting the tests, as we explored in [23], the frameworks could be modified to allow instances to notify each other when a defect is discovered, so that other instances can try to reproduce the failed test, which would further aid in fault localization. In some applications, for instance scientific calculations, it may also be desirable for the system to notify the user that a defect may have been detected, and that the results of the calculation may not be completely accurate.

9.3 Conclusion

In this thesis I will explore approaches and frameworks for testing software applications for which there is no reliable test oracle. In some cases, developers' knowledge of the software's properties is used to create tests that are executed "from within", but do not affect the state of the executing program. In other cases, properties of the entire application can easily be specified so that system testing can be performed while the program runs. The thesis will demonstrate that such approaches are feasible for revealing defects that could not otherwise be discovered (or could not easily be discovered) using current testing techniques.

Testing in the deployment environment and addressing the testing of applications without oracles have been identified as two of the future challenges for the software testing community [8]. As applications

that fall into this category - such as those in the domain of machine learning - become more and more prevalent and mission-critical, ensuring their quality and reliability gains the utmost importance.

9.4 Acknowledgments

In addition to thanking my committee (Gail Kaiser, Sal Stolfo, and Junfeng Yang), I would like to thank Rean Griffith, Phil Gross, Janak Parekh, and the members of the FSE-16 Doctoral Symposium program committee for their guidance and assistance in this proposal. Thanks also to Lori Clarke and Lee Osterweil for their suggestions regarding the development of the runtime testing approaches, and to T.Y. Chen for his advice on using metamorphic testing. Matt Chu, Lifeng Hu, Kuang Shen, Del Slane, and Ian Vo all were instrumental in designing and implementing the Invite, Corduroy, Amsterdam, and Columbus testing frameworks.

10 Appendix A - Implementation of Corduroy and Extension to JML

In [72], we describe an extension to the Java Modeling Language (JML) [49] that allows for the specification of metamorphic properties. These properties would be read by a pre-processor we developed called *Corduroy*, which would create test methods that could then be executed using JML runtime assertion checking [12]. This section summarizes the Corduroy implementation details. Although Corduroy is not part of the thesis work, the Columbus frameworks will build on it conceptually by providing similar mechanisms for specifying and executing the metamorphic checks at runtime.

As is customary in JML, a function's properties are specified in annotations in the comments preceding the method with which they are associated, or in a separate file. In our extension, the metamorphic properties are specified in a line starting with the tag "@meta" and then are followed by a Java boolean expression that states the property.

```
/*@
 @meta \result == sine(x + 2 * Math.PI);
 @meta \result == -1 * sine(-x);
 */
 public double sine (double x) { ... }
```

Figure 6: Example of specification of metamorphic properties for sine using JML

Figure 6 shows a basic example for the sine function. It uses the metamorphic properties sin(α) = sin(α + 2π) and sin(α) = -sin(-α). Note that, assuming the method returns a non-void value, the JML keyword "\result" can be used to represent the method's return value when specifying the metamorphic property, which is to be checked after the function has completed, so that the function need not be called again with the original input.

In this particular example, the property could in fact be specified without any modification to JML (using the "@ensures" keyword to specify it as a post-condition), but only if the function is pure, *i.e.*, has no side effects. Our extension to JML not only allows for the inclusion of functions that have limited side effects (by restoring some parts of the state after the properties have been checked), but also adds additional syntax and built-in functions that facilitate the specification of metamorphic properties.

As it is written, the above example may fail even if the function is working correctly, due to imprecision in Java's floating point calculations. For instance, the Math.sin function computes the sine of 6.02 radians and the sine of (6.02 + 2 * Math.PI) radians as having a difference of 7 * 10^{-15}, which in most applications is probably close enough, but is not exactly the same when compared using double-equals in Java, which would lead to a false positive in many cases. In order to simplify the specification of the metamorphic properties, our extension to JML allows floating point values to be compared using a built-in tolerance level, and the comparison returns true if the values are within that tolerance. Of course, if developers want finer control over the tolerance, they can explicitly take the absolute value of the difference and then comparing it to a tolerance, as is customary in JML.

To simplify the specification of some of the types of metamorphic properties that we feel would be typical, based on our evaluation in [71], we have also added special keywords to the JML syntax, using

the JML style of starting keywords and operators with a backslash. These allow for the execution of operations on arrays (of Objects or primitives) or on classes that implement the Java Collection interface that would used during the test; Table 3 explains these built-in functions.

\add(A , c)	Adds a constant c to each element in array or Collection A
\multiply(A , c)	Multiplies each element in array or Collection A by a constant c
\shuffle(A)	Randomly permutes the order of the elements in array or Collection A
\reverse(A)	Reverses the order of the elements in array or Collection A
\negate(A)	If the elements in A are numeric, multiplies each by -1
\include(A , x)	Inserts an element x into array A
\exclude(A , x)	Removes an element x from array A

Table 3: Additional keywords added to JML for manipulating arrays

An example of the use of these keywords appears in Figure 7. When calculating the standard deviation for an array of integers, shuffling the values should not affect the result, since the calculation does not depend on the initial ordering of the elements. However, multiplying each element by 2 is expected to double the calculated standard deviation.

```
/*@
 @meta \result == standardDev(\shuffle( A ));
 @meta \result * 2 == standardDev(\multiply( A , 2));
 */
 public double standardDev (int[] A) { ...  }
```

Figure 7: Example of using built-in array functions for specifying metamorphic properties

Some metamorphic properties may only hold under certain conditions or certain values for the input, for example if the input is positive or non-null. We allow for the inclusion of conditional statements when specifiying metamorphic properties, using if/else notation as opposed to the question mark-colon notation currently supported in JML. Figure 8 shows an example.

```
/*@
 @meta if (A != null && A.length > 0)
      average(\multiply(A, 2)) == 2 * \result;
 */
 public double average (double[] A) { ...  }
```

Figure 8: Conditional metamorphic property

Functions that are non-deterministic may still have metamorphic properties, though these would be 1-to-many relationships of inputs to possible outputs, rather than 1-to-1 mappings as we have discussed so far. For instance, a function that solves a quadratic equation may return the two possible values in an array, where either $[x_1, x_2]$ or $[x_2, x_1]$ is correct. Thus, the metamorphic property would need

to check that the new output is equal to one of these two possibilities. In other cases, the new output might be expected to fall within some range of numbers. To make these properties easier to express, we add two additional boolean functions, as described in Table 4.

Consider, for example, a function in a personal finance application that simulates market conditions and predicts the value of the user's portfolio after a certain amount of time. This function may use a Monte Carlo algorithm that uses randomness to simulate many possibilities and then reports the average as its output. If the value of each holding in the portfolio is doubled, we cannot expect that the predicted value will exactly be doubled, since the function is non-deterministic, but we may be able to specify that the value should not be *less* than the original output, and perhaps should not be more than four times that value. Thus, we can specify this metamorphic property as demonstrated in Figure 9.

\in { x ; S }	Returns true if the value x is equal to a member of set S
\inrange { x ; x_1 ; x_2 }	Returns true if x >= x_1 and x <= x_2

Table 4: Additional keywords for handling non-determinism in specifications

```
/*@
 @meta \inrange { predict(\multiply(holdings, 2) ;
           \result ; \result * 4 };
 */
 public double predict (ArrayList holdings) { ...  }
```

Figure 9: Example of metamorphic properties specifying a range of values

Although some of these metamorphic properties can be expressed in JML using boolean operators (such as logical AND and OR) within the relationship specification, these extensions should make the notation simpler and easier to understand, and reduce the chance of incorrectly specifying the metamorphic relationship.

Rather than modify or extend any existing JML implementation, Corduroy acts as a pre-processor that converts the specification of metamorphic properties into corresponding test functions and pure JML specifications, so that the code can then be compiled by any JML-compliant tool. When the code is executed, if runtime assertion checking is enabled in the virtual machine, the JML specifications will invoke the test functions, and the metamorphic properties can then be evaluated.

```
/*@
 @meta \result * \result ==
    principal * calcInterest(P, r, n, 2 * t);
 @assignable balance;
 */
 public double calcInterest (double P, double r,
    double n, double t) { ...  }
```

Figure 10: Example of metamorphic property expressed by extended JML specification.

```
/*@
@meta \result * \result ==
    principal * calcInterest(P, r, n, 2 * t);
@assignable intVariable;
@ensures metaTestCalcInterest(P, r, n, t, \result) == true;
*/
public double calcInterest (double P, double r,
    double n, double t) { ...  }

protected synchronized boolean metaTestCalcInterest
    (double P, double r, double n, double t, double result)
 {
  double __balance = balance;
  try {
    if ((result * result == principal *
     calcInterest(P, r, n, 2 * t)) == false)
      return false;
    return true;
  }
  catch (Exception e) {
    return false;
  }
  finally {
    balance = __balance;
  }
 }
```

Figure 11: Example of specification from Figure 10 after processing by Corduroy.

Each metamorphic property as specified for the original method is translated into valid Java that checks that the boolean expression is true. To support the keywords added to JML by our approach, we use calls to a built-in Corduroy library of static methods. Each property is checked individually, and if an expression returns false, the test method returns false immediately. This means that the "ensures" clause in the original method will fail, and if runtime assertion checking is enabled, the JML runtime environment will handle it accordingly.

As an example, consider a function in a BankAccount class that calculates compound interest, but as a side effect also updates the BankAccount's balance. One of its metamorphic properties is that if the amount of time is doubled and the value is multiplied by the principal, the result will be equal to the square of the original amount of interest. Figure 10 demonstrates an example of the specification written using our extension to JML; Figure 11 shows the code after it has been processed by Corduroy.

Note that our approach intentionally does not dictate what action the application should take if a defect is discovered through the testing, *i.e.*, if the post-condition assertion check fails. Rather, this is handled by the JML runtime environment, and would typically result in an exception being thrown; the stack trace of the exception would then indicate which metamorphic property did not hold.

11 Appendix B - In Vivo Testing and the Invite Framework

In [70], we developed an implementation of the In Vivo Testing framework for Java applications called *Invite*. This framework was developed using the aspect-oriented programming language AspectJ [2]. Here we describe the details of the implementation, the result of feasibility studies, and some experiments to measure performance overhead. Although In Vivo Testing itself is not part of the thesis work, the concept of Metamorphic Runtime Checking and the proposed Columbus framework are conceptually based on the approach.

11.1 Implementation

Whenever a method of an instrumented class is invoked, the framework uses the percentage value ρ for that method to decide whether to execute a test. If Invite decides that a test is to be run, it uses Java Reflection to see if the method has a corresponding "test" method (for performance reasons, however, we cache the results of previous checks to see if the test method exists). This is the In Vivo test that will then be executed. If the method being executed is also associated with existing unit tests (and not In Vivo tests), one of its corresponding unit tests may be executed instead. The purpose of running a method's corresponding test method is so that the test is executed at the same point in the program (the same state) as the method itself. This makes it possible to see how the test performs in the same state in which the method performs, which is preferable to arbitrarily choosing a random test to execute, since there may be states when such a test is *not* expected to work correctly.

If a test method exists and it is determined that a test should be run, Invite then forks a new process (which is a copy of the original) to create a sandbox in which to run the test code, ensuring that any modification to the local process state caused by the unit test will not affect the "real" application, since the test is being executed in a separate process with separate memory. As Invite is currently implemented in Java, and there is no "fork" in Java, we have used a JNI call to a simple native C program which executes the fork. Performing a fork creates a copy-on-write version of the original process, so that the process running the unit test has its own writable memory area and cannot affect the in-process memory of the original. Once the test is invoked, the application can continue its normal execution, while the unit test runs in the other process. Note that the application and the unit test run in parallel in two processes; the test does not pre-empt or block normal operation of the application after the fork is performed.

In the current implementation of Invite, unit test modifications to network I/O, the operating system, external databases, *etc.* are not automatically undone; the sandbox only includes the in-process memory of the application (through the copy-on-write forking). To address this, we implemented a prototype of a modified JDK, built on the open source OpenJDK [78], so that files are not modified by the tests either. Though this somewhat limits the type of testing that can be performed currently, there are still many categories of defects that can be detected when considering tests that only utilize and affect the state of the process in memory. Furthermore, if a "tearDown" method exists in the class in which the test was run, that method is executed upon completion of the test, allowing for any programmatic clean-up that needs to be done (though, as described previously, it is not necessary to restore in-process memory to its original state, only that of external systems).

When the unit test is completed, Invite logs whether or not it passed, and the process in which the unit test was run is terminated. Invite provides a tool for analyzing the log file and providing simple statis-

tics like the number of tests run, the number that passed/failed, and a summary of the success/failure of each instrumented method's unit test. We have also implemented a "client-server" version of Invite [23] in which all errors are reported back to a central server (presumably this would be set up at the vendor's location), and could be processed as in [80] or [62], wherein configuration parameters (like the frequency of test execution or even the list of classes to test) could then be modified.

We have also considered other policies for determining how frequently unit tests should be run, aside from the static configuration value. For instance, if it is desirable to have all the test cases run equally often, then the ρ value could be automatically adjusted to increase probability for a method that, empirically, runs rarely, and lowered for one that runs often. Another policy would be to multiply the weighting (which treats all essentially equally but considers how often they run in practice) by some factor that is larger for methods/classes where more bugs were found during lab testing and/or more field bugs were reported, so as to increase the likelihood of finding a bug in a potentially flawed method or class. Another solution may be to use a tool like the GAMMA system [79] [80] for determining which tests should be run under different circumstances, such as system load. The relative effects of these different policies are outside the scope of this paper.

11.2 Addressing Performance Concerns

The performance impact of such an approach is an immediate concern, and we have taken a number of steps to ensure that the system is feasible for use without causing significant slowdown of the software under test.

The first and perhaps most obvious measure we took is to allow the administrator to limit the number of simultaneous tests that are being executed, so that test processes are not created so frequently as to flood the CPU. When the maximum number of test processes are executing, the Invite framework is temporarily disabled so that no more tests are started. This gives the administrator a mechanism for keeping the number of processes under control.

The maximum allowed number of simultaneous test processes would ideally be less than or equal to the number of CPUs/cores in the machine. To take advantage of multiprocessor/multicore architectures, it is possible to configure Invite so that each process runs independently and does not interrupt the others. Each process is assigned to a separate CPU/core using an affinity setting (this is not supported in Java but is possible through a JNI call), thus ensuring that the tests do not run on the same CPU/core as the main process and limiting the overall impact on the application.

We have also investigated ways to reduce the overhead by distributing the testing load across multiple instances of the application under test. In [23], we discovered that it is possible to share the testing load across a small "application community" [54] in a software monoculture. We have also started to implement a mode that allows Invite to automatically alter the ρ value based on the desired frequency of test execution, or an acceptable performance overhead. For instance, the value can be raised when there is less usage of the application, so that more tests will run but the system will not be under excessive load, and load can be shared across different instances of the application.

11.3 Feasibility

To demonstrate the feasibility of the Invite testing framework, we applied it to OSCache 2.1.1 [81], which contained numerous known bugs that we speculated could be detected with the In Vivo Testing approach. Note that we did not explicitly address the problem of applications without test oracles in this work, but a caching tool is similar in that, as in machine learning applications, certain defects may not be obvious to the end user.

One of the bugs we discovered is that, under certain configurations, the method to remove an entry from the cache is unable to delete a disk-cached file if the cache is at full capacity.[3] A unit test that assumes an empty or new cache would pass, however; but when the cache is full, the In Vivo test would fail, revealing a bug that may not have been caught in the development environment. In another OSCache bug, setting the cache capacity programmatically does not override the initial capacity specified in a properties file when the value set programmatically is smaller.[4] A unit test for the method to set the cache capacity may assume a fixed value in the properties file and only execute tests in which it sets the cache capacity to something larger; this unit test would pass. However, if a system administrator sets the capacity to a large number in the properties file, an In Vivo test would fail when it tries to set the cache capacity to a smaller value, revealing the bug. In the last bug, flushing the cache, adding an item, and attempting to retrieve the item can occasionally result in an error, particularly if two calls to flush the cache happen within the same millisecond.[5] A unit test that tries this sequence of actions may simply never encounter the error by chance during testing in the development environment, but an application fitted with the In Vivo framework would catch it when it eventually occurs.

Unfortunately the unit tests that are distributed with that version of OSCache do not cover the methods in which those defects are found, so we created unit tests that would reasonably exercise those parts of the application. As expected, those tests passed in the development environment during traditional unit testing, primarily because we had created the tests assuming a clean state which we could control (which, we feel, is a reasonable and common assumption [44]). This took less than one hour to complete. We then developed In Vivo tests, using those unit tests as a starting point; it took less than one hour to complete this task. Although we did not have a real-world application based on OSCache for our testing, we created a driver that used the OSCache API to randomly add, retrieve, and remove elements of random size from a cache, and randomly flushed the cache. All three defects were revealed by Invite in less than two hours. The last to reveal itself was the one that only happened when the cache was at full capacity, which happened rarely in our test because the random adding, removing, and flushing did not allow it to reach capacity often; however, this defect may have revealed itself more quickly in a real-world application.

As mentioned previously, In Vivo Testing and the Invite framework are not part of the thesis work, and we will need to create a new framework (Columbus) that particularly allows for metamorphic testing. The work described here is meant to demonstrate the feasibility of the approach and of using a framework that conducts tests in the runtime environment without affecting the state from the users' point of view.

[3] http://jira.opensymphony.com/browse/CACHE-236
[4] http://jira.opensymphony.com/browse/CACHE-158
[5] http://jira.opensymphony.com/browse/CACHE-175

11.4 Performance Overhead

We are concerned with the performance impact of our approach, particularly in instrumenting potentially numerous method calls (perhaps all of them) to act as points at which tests can be executed, as well as the overhead incurred by using forking to create a new process in which the test would be run. We conducted some performance tests to determine the additional overhead introduced by the Invite framework. Although we will not be using Invite for performing Metamorphic Runtime Checking, this section gives a rough idea of the performance impact that can be expected.

In our experiments, we measured the performance overhead during our testing of OSCache 2.1.1, using Java 1.6.0 on a Linux Ubuntu 2.7.1 server with a dual-core 3.0GHz CPU and 1 GB of memory. Only minimal background system processes were executing during our tests. We first executed the test in our environment without the In Vivo Testing framework attached, to determine a baseline. The test consisted of 100,000 random calls to add, retrieve, and remove items from a cache, as well as to flush the entire cache. The time to complete the benchmark with no Invite instrumentation was 1062ms.

We then instrumented the four appropriate methods and created simple In Vivo tests for each; we then set the probability of running a test to 0. In this case, we could measure the overhead of the instrumentation itself from the inserted AspectJ code, which still has to check that probability on each method call, since the instrumentation of the code is done at compile-time but the configuration is checked at run-time. In this case, though, we did not need to consider the forking of new processes or parallel execution of any test code, since Invite would never execute any tests. This time, the test completed in 1080ms (1.6% increase), which indicated very little impact overall and is consistent with the small overhead caused by calls to weaved-in AspectJ code [41].

Next we configured Invite so that the probability ρ of running a test (for each of the four methods) was set to 0.1% (this version of Invite did not include the ability to set an acceptable overhead threshold). To demonstrate the effects of more frequent testing, we then repeated the tests with larger ρ values; the results are shown in Table 5. All tests ran on a different core from the original process.

Table 5: Results of performance testing

Percent of methods that execute tests (ρ)	Total time (ms)	%diff	Number of tests	Tests per second
Baseline	1062	-	-	-
0%	1080	1.6	0	0
0.1%	1115	4.9	39	34.9
1%	1140	7.3	51	44.7
10%	1276	20.1	58	45.5
100%	1299	26.6	72	55.4

Note that the number of tests executed does not increase by an order of magnitude just because the percent probability of running a test does: depending on how long the test takes to run, and the allowed maximum number of concurrent tests, only a certain number of tests could be fit in before the program runs to completion. It is possible that an In Vivo test that takes a very long time to run will reduce the

overall number of tests run, but it will also reduce the overhead (since new tests are not starting), and we expect that the ratio of overhead to number of tests run would stay about the same even in those cases.

We also ran similar experiments on a quad-core machine, in which we allowed for two and then three simultaneous tests to be run. The results were as expected: the performance overhead increased because more test processes were being forked, but the number of tests run during the experiment also increased with the number of allowable simultaneous tests. See [70] for more details.

We have previously investigated a distributed approach to In Vivo Testing [23] and demonstrated that by reducing the number of tests each instance of an application runs, but by having multiple instances of the application conduct tests, the global number of tests is constant but the performance overhead on each instance is reduced. We have also looked into ways in which a small application community can balance testing load by profiling each instance (*i.e.*, the distribution of the function executions) and then calculating an assignment of test cases such that each instance has the same expected testing overhead. These particular techniques, however, are out of scope for this thesis.

References

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

[2] AspectJ. http://www.eclipse.org/aspectj/.

[3] C. Atkinson and H.-G. Gross. Built-in contract testing in model-driven, component-based development. In *Proc. of ICSR Workshop on Component-Based Development Processes*, 2002.

[4] G. K. Baah, A. Gray, and M.J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *Proc. of the 3rd International Workshop on Software Quality Assurance*, pages 70–77, 2006.

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

[6] F. Barbier and N. Belloir. Component behavior prediction and monitoring through built-in test. In *Proc. of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 17–12, April 2003.

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

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

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

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

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

[12] A. Bhorkar. A run-time assertion checker for Java using JML. Technical Report TR00-08, Dept. of Computer Science, Iowa State Univ., 2000.

[13] D. Brenner and C Atkinson et al. Reducing verification effort in component-based software engineering through built-in testing. *Information System Frontiers*, 9(2-3):151–162, 2007.

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

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc of the seventh international conference on World Wide Web*, pages 107–117, April 1998.

[16] A. Califano, A. Floratos, M. Kustagi, and J. Watkinson. geWorkbench: An Open-Source Platform for Integrated Genomics. http://www.geworkbench.org/.

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

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

[19] T. Y. Chen, S. C. Cheung, and S.M. 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.

[20] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *Proc. of the International Workshop on Software Technology and Engineering Practice (STEP)*, pages 94–100, 2004.

[21] 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.

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

[23] M. Chu, C. Murphy, and G. Kaiser. Distributed in vivo testing of software applications. In *Proc. of the First International Conference on Software Testing, Verification and Validation*, April 2008.

[24] 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.

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

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

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

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

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

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

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

[32] 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.

[33] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

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

[35] R. Griffith and G. Kaiser. Adding self-healing capabilities to the common language runtime. Technical Report CUCS-005-05, Dept. of Computer Science, Columbia University, January 2005.

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

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

[38] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, 2002.

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

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

[41] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on aspect-oriented software development (AOSD)*, pages 26–35, 2004.

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

[43] T. Joachims. *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

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

[45] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know. In *Proc of the 10th International Software Metrics Symposium*, 2004.

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

[47] A. Krishna et al. A distributed continuous quality assurance process to manage variability in performance-intensive software. In *19th ACM OOPSLA Workshop on Component and Middleware Performance*, 2004.

[48] J.-C. Laprie and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[49] 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.

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

[51] SVM Application List. http://www.clopinet.com/isabelle/Projects/SVM/applist.html.

[52] C. Liu and D. Richardson. Software components with retrospectors. In *Proc. of International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 63–68, June 1998.

[53] C. Liu and D. J. Richardson. RAIC: Architecting dependable systems through redundancy and just-in-time testing. In *ICSE Workshop on Architecting Dependable Systems (WADS)*, 2002.

[54] M. E. Locasto, S. Sidiroglou, and A.D. Keromytis. Software self-healing using collaborative application communities. In *Proc. of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, pages 95–106, February 2006.

[55] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, 2006.

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

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

[58] C. Mao. AOP-based testability improvement for component-based software. In *31st Annual International COMPSAC, vol. 2*, pages 547–552, July 2007.

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

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

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

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

[63] M. Merdes et al. Ubiquitous RATs: how resource-aware run-time tests can improve ubiquitous software systems. In *Proc. of the 6th International Workshop on Software Engineering and Middleware*, pages 55–62, 2006.

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

[65] M. Middendorf, A. Kundaje, M. Shah, Y. Freund, C. H. Wiggins, and C. Leslie. Motif discovery through predictive modeling of gene regulation. *Research in Computational Molecular Biology*, pages 538–552, 2005.

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

[67] M. Momotko and L. Zalewska. Component+ built-in testing: A technology for testing software components. *Foundations of Computing and Decision Sciences*, 29(1-2):133–148, 2004.

[68] C. Murphy. Using runtime testing to detect defects in applications without test oracles. In *Proc of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008.

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

[70] C. Murphy, G. Kaiser, M. Chu, and I. Vo. Quality assurance of software applications using the in vivo testing approach. Technical Report cucs-045-08, Dept. of Computer Science, Columbia University, 2008.

[71] 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.

[72] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. Technical Report cucs-044-08, Dept. of Computer Science, Columbia University, 2008.

[73] L. Naslavsky and R.S. Silva Filho et al. Distributed expectation-driven residual testing. In *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04)*, 2004.

[74] L. Naslavsky et al. Multiply-deployed residual testing at the object level. In *Proc. of IASTED International Conference on Software Engineering (SE2004)*, 2004.

[75] 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.

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

[77] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In *In Proceedings of ICFEM06, volume 4260 of LNCS*, pages 494–513. Springer-Verlag, 2006.

[78] OpenJDK. http://openjdk.java.net/.

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

[80] A. Orso, D. Liang, and M.J. Harrold. Gamma system: Continuous evolution of software after deployment. In *Proc. of ISSTA 2002*, pages 65–69, 2002.

[81] OSCache. http://www.opensymphony.com/oscache.

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

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

[84] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st ICSE*, pages 277–284, May 1999.

[85] I. Pavlova, M. kerholm, and J. Fredriksson. Application of built-in-testing in component-based embedded systems. In *Proc. of the 2006 ISSTA Workshop on the Role of Software Architecture for Testing and Analysis*, pages 51–52, 2006.

[86] RapidMiner. http://rapid-i.com/.

[87] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing project. http://www.ics.uci.edu/~djr/edcs/PerpTest.html.

[88] D. Rubenstein, L. Osterweil, and S. Zilberstein. An anytime approach to analyzing software systems. In *Proc. of the 10th International FLAIRS Conference (Florida Artificial Intelligence Research Society)*, pages 386–391, May 1997.

[89] D. Saff and M.D. Ernst. Reducing wasted development time via continuous testing. In *Proc. of the 14th International Symposium on Software Reliability Engineering*, page 281, 2003.

[90] D. Saff and M.D. Ernst. An experimental evaluation of continuous testing during development. In *Proc. of ISSTA 2004*, pages 76–85, 2004.

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

[92] 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.

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

[94] R. Servedio. Personal communication, 2006.

[95] Y. Song, S. J. Stolfo, and A. D. Keromytis. Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic. In *Proc of the 16th Annual Network and Distributed System Security Symposium (NDSS),*, February 2009.

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

[97] J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component-based software systems. *Software Quality Journal*, 10(2), September 2002.

[98] K. Wang, J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. of the Ninth International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[99] 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.

[100] Y. Wang et al. On built-in test reuse in object-oriented framework design. *ACM Computing Surveys*, 32(1), March 2000.

[101] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Software Eng., SE-12*, pages 1128–1138, December 1986.

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

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

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

[105] M. Young. Perpetual testing. Technical Report AFRL-IF-RS-TR-2003-32, Univ. of Oregon, February 2003.

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

[107] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.