Metamorphic Testing Techniques to Detect Defects in Applications without Test Oracles

Christian Murphy

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2010

Abstract

Metamorphic Testing Techniques to Detect Defects in Applications without Test Oracles

Christian Murphy

Applications in the fields of scientific computing, simulation, optimization, machine learning, etc. are sometimes said to be "non-testable programs" because there is no reliable test oracle to indicate what the correct output should be for arbitrary input. In some cases, it may be impossible to know the program's correct output *a priori*; in other cases, the creation of an oracle may simply be too hard. These applications typically fall into a category of software that Weyuker describes as "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known." The absence of a test oracle clearly presents a challenge when it comes to detecting subtle errors, faults, defects or anomalies in software in these domains.

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

Without a test oracle, it is impossible to know in general what the expected output

should be for a given input, but it may be possible to predict how changes to the input should effect changes in the output, and thus identify expected relations among a set of inputs and among the set of their respective outputs. This approach, introduced by Chen et al., is known as "metamorphic testing". In metamorphic testing, if test case input x produces an output $f(x)$, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation function t, which can then be applied to the input to produce $t(x)$; this transformation then allows us to predict the expected output $f(t(x))$, based on the (already known) value of $f(x)$. If the new output is as expected, it is not necessarily right, but any violation of the property indicates that one (or both) of the outputs is wrong. That is, though it may not be possible to know whether an output is correct, we can at least tell whether an output is *incorrect*.

This thesis investigates three hypotheses. First, I claim that an automated approach to metamorphic testing will advance the state of the art in detecting defects in programs without test oracles, particularly in the domains of machine learning, simulation, and optimization. To demonstrate this, I describe a tool for test automation, and present the results of new empirical studies comparing the effectiveness of metamorphic testing to that of other techniques for testing applications that do not have an oracle. Second, I claim that conducting function-level metamorphic testing in the context of a running application will reveal defects not found by metamorphic testing using system-level properties alone, and introduce and evaluate a new testing technique called Metamorphic Runtime Checking. Third, I hypothesize that it is feasible to continue this type of testing in the deployment environment (i.e., after the software is released), with minimal impact on the user, and describe an approach called In Vivo Testing.

Additionally, this thesis presents guidelines for identifying metamorphic properties, explains how metamorphic testing fits into the software development process, and discusses suggestions for both practitioners and researchers who need to test software without the help of a test oracle.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Gail Kaiser, who has provided me invaluable guidance and wisdom in all matters related to my research and publishing. I truly appreciate the support, and am thankful for the flexibility I've been allowed in choosing projects and research directions. I am certainly a better academic because of her efforts.

I would also like to thank Lee Osterweil and Lori Clarke for their advice in various matters related to software testing research, and to T.Y. Chen for sharing his insights and wisdom into how metamorphic testing can be employed, improved, and evaluated. I also extend my thanks to the other members of my thesis committee, Sal Stolfo and Junfeng Yang, who both encouraged me to consider the practical applications of my work.

I would be remiss if I did not especially thank Adam Cannon, who acted as a mentor to me throughout my graduate school career, and who motivated me to improve myself as a scholar, educator, and person.

The work described in this thesis has been supported by the Programming Systems Lab, funded in part by NSF grants CNS-0905246, CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH grant 1U54CA121852-01A1. Various members (past and present) of the Programming Systems Lab gave me substantial advice and assistance as I put together this document, including Phil Gross, Janak Parekh, Suhit Gupta, Rean Griffith, Hila Becker, Swapneel Sheth, Leon Wu, and Nipun Arora. I have also had the pleasure of supervising a number of graduate and undergraduate students as part of this work, and hope they gained as much from the experience as I did: Lifeng Hu, Kenny Shen, Ian Vo, Matt Chu, Huning Dai, Moses Vaughan, Waseem Ilahi, Sahar Hasan, and Chad Brauze.

I would like to thank the developers of the systems that we tested in our various experiments for helping us get started and for putting up with my questions: Marta Arias, Sandy Wise, and M.S. Raunak. I also thank Xiaoyuan Xie for her efforts in our joint collaborations on metamorphic testing and empirical studies.

I absolutely must give the utmost thanks to my parents, Thomas Murphy and Jane Neapolitan, and to my brother, Andy Murphy, for accepting the fact that I refuse to grow up.

And last but by no means least, I thank my wife, Ina Choi. Every single one of my accomplishments can be directly attributed to her emotional support and encouragement.

To my dad and to Ina,
who have always been there for me,
even from far away.

Chapter 1

Introduction

Assuring the quality of applications such as those in the fields of scientific computing, simulation, optimization, 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. These applications are sometimes referred to as "non-testable programs" [190], and 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"* [190].

Machine learning applications fall into this category of programs without test oracles as well. Such applications are developed to discover previously-unknown trends in or relationships among large sets of data; if there were some way of knowing these results in advance, there would be no need to develop the software in the first place. As these types of applications become more and more prevalent in society [125], ensuring their quality becomes more and more crucial. For instance, there are over fifty different real-world applications [178], ranging from facial recognition to computational biology, that use the Support Vector Machines [183] machine learning classification algorithm alone.

Additionally, machine learning ranking applications are widely used by Internet search engines [25], and intrusion detection systems that use machine learning algorithms are clearly becoming more important as critical data is stored online and attackers seek to access it or gain control of systems [86].

One emerging domain of machine learning applications is in the area of clinical diagnosis, using a combination of systems-level biomolecular data (e.g., microarrays or sequencing data) and conventional pathology tests (e.g., blood count, histological images, and clinical symptoms). It has been demonstrated that a machine learning approach of multiple data types can yield more objective and accurate diagnostic and prognostic information than conventional clinical approaches alone [82]. However, for clinical adoption of this approach, these programs that implement machine learning algorithms must be rigorously tested to ensure their quality and reliability. A mis-diagnosis due to a software fault can lead to serious, even fatal, consequences.

In all of these fields, formal proofs of an algorithm's optimal quality are not sufficient to guarantee that an *implementation* uses that algorithm correctly and is free of errors. It is certainly possible that a programmer may accidentally insert a defect into the code during implementation, for instance incorrectly performing a calculation or causing an off-by-one error, i.e., forgetting to properly adjust a variable by one. In 1994, Hatton and Roberts pointed out a "disturbing" number of calculation errors in software used in the earth sciences community [79], and - even worse - followed that up with an article 13 years later in which Hatton pointed out that things were not getting any better, claiming that *"many scientific results are corrupted, perhaps fatally so, by undiscovered mistakes in the software used to calculate and present those results"* [78]. More recently, others have demonstrated that software engineering techniques are generally not prevalent amongst programmers in the scientific community, and that although scientists understand the importance of software testing, few feel that they have a solid understanding of how to sufficiently test the programs that they create [77].

This thesis investigates solutions to the problem of testing applications that do not have test oracles. Given the importance of software in domains like machine learning, and given the apparent lack of quality in some areas of scientific computing, and given the general difficulty of testing applications without test oracles, the challenge and significance of this research is clear. This thesis describes enhancements to existing approaches, introduces novel techniques, and presents new empirical studies that demonstrate advancement in the state of the art. This work also addresses the testing of non-deterministic applications that do not have test oracles, which introduces additional challenges since multiple invocations of the code under test may yield different results. Last, this thesis generalizes the proposed techniques so that they can be used effectively in a variety of application domains, and provides guidelines to researchers and practitioners who need to test applications without the help of a test oracle.

1.1 Definitions

Before we further discuss the problem statement, requirements, and proposed approach, this section first formalizes some of the terms used throughout this thesis.

- A **defect**, also referred to as an "error" or "bug", is the deviation of system external state from correct service state [97]. See Section 1.4 below for a description of the types of defects specifically investigated in this thesis.

- A **fault** is the adjudged or hypothesized cause of a defect [97].

- 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 functionality [97].

- 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) [15]. This is sometimes referred to as a "comparison-based test oracle" [151].

- The **development phase** includes all activities from presentation of the stakeholders' initial concept to the decision that the system has passed all acceptance tests and is ready to deliver service in its users' environment(s) [97].

- 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 [97].

- A **deployment environment**, or use environment, refers to a setting in which software is 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 through its interfaces; and the physical infrastructure that supports such activities [97].

- 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.

- **Metamorphic testing** is a technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure. It is a methodology of reusing input test data to create additional test cases whose outputs can be predicted [32]. This is not to be confused with other uses of the word "metamorphic" in computer science, such as metamorphic code, metamorphic viruses, etc.

- In this thesis, we refer to a **function** as a clearly delineated group of instructions that perform a specific task [63]. Depending on the parlance of the particular programming

language, these may also be referred to as “methods”, “procedures”, “subroutines”, etc. We do not mean a function in the mathematical sense, but rather in the computer programming sense.

1.2 Problem Statement

Independent of advances in software testing, there still remains a certain class of applications that have no reliable test oracle to indicate what the correct output should be for arbitrary input. Regardless of whether there is an oracle, we cannot in general demonstrate correctness of the implementation, i.e., the absence of defects, but we need a testing approach that can at least demonstrate the *presence* of defects in such applications. Such defects may occur at the function 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. These “partial oracles” [151] are typically only useful for very simple inputs, however, and may not have much power at revealing defects [127, 190]. Additionally, other inputs, such as those that push boundary or timing limits, can be used to reveal gross errors, e.g., catastrophic failures (crashes) or infinite loops. However, an approach is needed that will reveal more *subtle* defects for *general* input, as opposed to obvious defects for a limited set of input.

Testing techniques typically require some set of test input data, which could be generated using various techniques, such as equivalence partitioning [135] or random testing [73]. However, inputs chosen using these techniques might not consider a sufficient variety of potential inputs and subsequent system states [129, 130]. Some defects in such systems (whether they have a test oracle or not) may only be found under certain application states that may not have been tested because of the infeasibility of exhaustive testing. Thus, a strategy that specifically considers these states and these inputs would likely be more effective in revealing defects.

1.3 Requirements

This thesis seeks 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 to the problems described above should meet the following requirements.

1. **Indicate defects in applications without test oracles**. Although it may be impossible in the general case to indicate that an output is *correct* for the given input, the solution must at least be able to indicate whether the output is *incorrect* for at least some particular cases.

2. **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. For instance, if the software crashes, hangs, or produces output that is clearly incorrect, virtually any existing technique would be able to detect the error. We require an approach that can reveal defects in output that may "look" correct, but actually is not.

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 computing, 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.

4. **Be configurable**. The software tester 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, at what points during the program execution the tests are to be run, what to do if a test fails, the acceptable overhead, etc.

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

familiar or easy-to-learn techniques. Aside from specifying test cases and setting up the testing framework, the process should be completely automated and not rely on manual intervention.

6. **Be efficient**. The user of a system that is conducting tests during execution should not observe any noticeable performance degradation or any side effects, such as test outputs being displayed on the screen or in a file.

1.4 Scope

Although we present a solution that is designed to be general purpose and applicable to a variety of applications, in this thesis we specifically limit our scope to applications that do not have test oracles, primarily in the domains of machine learning, discrete event simulation, and optimization.

Additionally, as pointed out in Section 1.1, in this thesis a "defect" is considered to be a deviation between the system's expected state and its actual state, i.e., when an implementation differs from its specification. A defect may also be defined as the violation of a sound property of the software. Note that in this thesis, a defect does *not* refer to errors that come about due to floating point imprecision or rounding, though these types of errors may be detectable using the approaches defined here; nor does it refer to errors in assumptions about the accuracy with which a scientific model represents a real-world process or phenomenon, which we would consider to be errors in the algorithm, not in the implementation.

It is true that software crashes, security vulnerabilities, concurrency bugs, and timing errors fit this definition of "defect". However, this thesis focuses only on errors for which the program terminates normally and produces an output, but the output is incorrect due to discrete localized defects that are the result of a specific (human) programming error at a specific point in the code, such as off-by-one errors, errors in calculation (adding instead of

subtracting, etc.), or errors in logic (using "and" instead of "or", etc.). These are the types of defects that are particularly hard to detect in the applications in the domains of interest [79].

1.5 Proposed Approach

One popular technique for testing applications that do not have test oracles is to use a "pseudo-oracle" [50], in which multiple implementations of an algorithm process the same input and the results are compared; if the results are not the same, then at least one of the implementations contains a defect. This approach is not always feasible, though, since multiple implementations simply may not exist, or may be too complex to develop. Others have pointed out that, in practice, the implementations may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [93], or that the act of comparing the outputs may be error-prone if the implementations can arrive at different results without actually having defects [24].

However, even without multiple implementations, often applications without test oracles exhibit properties such that **if an input produces an output, and the input is then modified in a certain way, it may be possible to predict its new output, given the original output**. If the application does exhibit such a property and the new output is as expected, that does not necessarily mean that the implementation is working correctly. However, if the property is violated, and the output is not as expected, then there *is* a defect. That is, the application acts as a pseudo-oracle for itself. This approach was introduced by Chen et al. and is known as "metamorphic testing" [32].

Metamorphic testing is 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 test case input x produces an output $f(x)$, the so-called "metamorphic properties" of the

code under test can then be used to guide the creation of a transformation function[1] 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 and the property is sound, then a defect must exist. Of course, adhering to this property does not necessarily mean that the output is right (i.e., even if the outputs are as expected, both $f(x)$ and $f(t(x))$ could still be wrong), but although we cannot know whether the output is correct, we may at least be able to tell that it is *incorrect*.

Metamorphic testing has previously been shown to be applicable to software for which there is no test oracle [37, 202]. In some cases, these works have looked at cases in which there cannot be an oracle for a particular application because the output cannot be known in advance [38]; in others, the work has considered the situation in which the oracle is simply absent or difficult to implement [28].

In practice, metamorphic testing can be a manually intensive technique for all but the simplest cases. The transformation of input data can be laborious for large data sets, or nearly impossible for input that is not in human-readable format. Similarly, comparing the outputs can be error-prone for large result sets, especially when slight variations in the results are not actually indicative of errors (i.e., false positives). Moreover, non-deterministic applications introduce further complications in the comparing of outputs. Thus, to reduce the likelihood of human error and increase testers' productivity, part of our approach involves **automating the process of conducting metamorphic testing**.

In addition to recommending the use of metamorphic testing to address this so-called "oracle problem", we also suggest that **we can detect additional defects by conducting metamorphic testing at the function level from within the running application**. That is, as opposed to performing system testing based solely on metamorphic properties of the entire application, or by conducting unit testing of isolated pieces of code, we suggest testing applications that do not have test oracles by checking the metamorphic properties

[1] Note that by "function", we refer to a procedure, subroutine, method, etc.

of some of their individual functions as the full application runs, instead of testing the functions in isolation, as is typical in unit testing. This approach requires the use of a new type of test framework 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 no matter what the functions inputs are. This would go beyond passive application monitoring (e.g., [145]) and actively test the application as it runs.

As the final part of our approach, we point out that **continuing to execute tests in the field, after deployment, can provide representative test data and may 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) may also be revealed.

Our solution, therefore, entails automating the process of metamorphic testing at both the system and the function level, and making it possible to check metamorphic properties as the software executes, even in the deployment environment. For either an individual function or for the entire application, a testing framework first captures the initial input/output pairs, which may be from test input data or from actual execution in the field. The framework then applies the function's or application's metamorphic properties to derive new test input, so that it should be possible to predict the corresponding test output. Although it cannot be known whether the outputs are correct, if they are not as predicted then a defect has been revealed. When this process is conducted in the field, the framework must ensure that users do not notice this testing, e.g., see the test output, experience a sudden performance lag, etc.

Of course, a solution to this problem does not necessarily need to be limited to applications without test oracles. Even applications that have test oracles also have properties (metamorphic or otherwise) that can be checked as the program executes, even while the program is running in the field. Some defects in such systems may only be found under certain application states that may not have been tested prior to deployment: for large,

complex software systems, it is typically impossible in terms of time and cost to reliably test all possible system states before releasing the product to the end users. Thus, we can generalize the solution so that arbitrary properties of the system or its constituent functions can be checked as the software runs in the deployment environment.

1.6 Hypotheses

The main hypotheses investigated are as follows:

1. **For programs that do not have a test oracle, an automated approach to metamorphic testing will advance the state of the art in detecting defects**. That is, we will show that the approach is more effective at revealing defects than other techniques such as using partial oracles or runtime assertion checking, and more practical than using pseudo-oracles, formal specifications, extrinsic interface contracts, or log/trace file analysis, all of which have been suggested as solutions for testing programs without oracles [15].

2. Furthermore, **an approach that conducts function-level metamorphic testing in the context of a running application will reveal defects not found by metamorphic testing using system-level properties alone**. We will show that checking the metamorphic properties of individual functions while the program is running can find defects that metamorphic testing based on properties of the entire application may not discover.

3. **It is feasible to continue this type of testing in the deployment environment, with minimal impact on the end user**. Such a solution would not modify the application state from the users' perspective, and would have acceptable performance overhead.

We primarily apply these approaches to applications in the domain of machine learning, 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 may be a result of untested deployment environments, configurations, etc. Although the focus is mostly on machine learning applications, we also show generalizability by applying the technique to applications in other domains that do not have test oracles, such as discrete event simulation, information retrieval, and optimization.

As these types of applications become more and more prevalent in various aspects of everyday life, it is clear that their quality and reliability take on increasing importance. This thesis advances the state of the art in testing these types of applications, and provides guidelines to developers and testers who are working with programs that have no test oracle.

1.7 Outline

The rest of this thesis is organized as follows:

- Chapter 2 motivates the work further and describes a set of **metamorphic testing guidelines** that can be followed to assist in the formulation and specification of metamorphic properties. It also discusses how metamorphic testing fits into the overall software development process.

- Chapter 3 describes an approach for 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 simplifies the process of conducting metamorphic testing, but also allows for metamorphic testing to be conducted in the production environment without affecting the user, so that real-world input can be used to drive the test cases. This chapter also introduces an implementation framework called **Amsterdam**, and discusses a new technique called **Heuristic Metamorphic Testing**, which is used for testing non-deterministic applications that do not have oracles, using application-specific heuristics. It also details the results of empirical studies that measure the approaches' effectiveness at detecting defects, and

quantitatively compares the approaches to other techniques such as pseudo-oracles, runtime assertion checking, and the use of a partial oracle.

- Chapter 4 introduces a new type of testing called **Metamorphic Runtime Checking**, which enhances metamorphic testing by making it possible to check the metamorphic properties of individual functions (as opposed to only those of the system as a whole) while the entire program is running. Metamorphic tests are executed at designated points in the program, within its current runtime context. We also present a system called **Columbus** that supports the execution of Metamorphic Runtime Checking from within the context of the running application. Columbus conducts the tests with acceptable performance overhead, and ensures that the execution of the tests does not affect the state of the original application process from the users' perspective.

- Chapter 5 generalizes the Metamorphic Runtime Checking approach into a technique called **In Vivo Testing**. This allows software to perform any type of test (not just metamorphic tests) on itself in the runtime environment, including unit or integration tests, or special "In Vivo tests" that are designed to check properties of the system regardless of its application state. We also present a framework called **Invite** that implements the approach.

- Chapter 6 presents related work for all three testing approaches. It also qualitatively compares the metamorphic testing approaches to other techniques for testing programs that have no oracle, such as using formal specifications, extrinsic interface contracts, and log/trace file analysis.

- Chapter 7 summarizes the main contributions of this work, discusses short-term and long-term future research directions, and concludes the thesis.

Chapter 2

Background

This chapter provides background and context for the work presented in this thesis. We begin in Section 2.1 by motivating the work through real-world examples of applications without test oracles. We then describe the metamorphic testing technique in further detail in Section 2.2, and discuss some previous work in the field in Section 2.3. In Section 2.4, we discuss guidelines for devising metamorphic properties, and also explain how the technique fits into the overall software development process. Last, in Section 2.5, we describe the applications of interest (from the domain of machine learning), and discuss their metamorphic properties, which will be used in the experiments in the following chapters.

2.1 Motivation

This line of research began with work in which we addressed the dependability of a machine learning (ML) application commissioned by a company for potential future experimental use in predicting impending electrical device failures in an urban power grid, using historic data of past failures as well as recent static and dynamic information about the devices [127]. Classification in the binary sense ("will fail" vs. "will not fail") is not sufficient because, after enough time, every device will eventually fail. Instead, a ranking of the

propensity of failure with respect to all other devices is more appropriate. The application uses a variety of ML algorithms in its implementation, and thus has no oracle to indicate whether it is producing the correct output, since if the correct output (the ranking) could be known in advance, then the application would not be needed. Although it is possible to know whether the application is *predicting* well by simply waiting a period of time and seeing whether the devices deemed most likely to fail actually do, that does not necessarily mean that the implementation is without defects; even random predictions could sometimes be right. Thus, software testing is necessary. The full application is detailed by Gross et al. [70], but is not otherwise discussed further here.

The dependability of the implementation of this system addresses real-world concerns, rather than just academic interest. Although it may be impossible to accurately predict all power outages (which can be due to weather, human error, hungry rats, etc.) there have been cases in which outages might be prevented via timely maintenance or replacement of devices that are likely to fail, such as the 2005 blackout in Java and Bali[1], and the 2008 blackout in Miami[2]. A dependable application in this domain may save money and even lives if it can accurately predict which devices are most likely to fail, so that preventative measures can be taken.

The impact of the research goes far beyond the particular application for which our investigations began. As machine learning applications become more and more prevalent in society [125], ensuring their quality becomes more and more crucial. There are over fifty different real-world applications [178], ranging from facial recognition to computational biology, that use implementations of the Support Vector Machines [183] machine learning classification algorithm. Additionally, ranking of search results is widely used by Internet search engines (e.g., [25]), also apparently using similar machine learning algorithms without test oracles. And other ML applications like those used for security and intrusion detection systems are clearly becoming more important as companies seek to protect crucial

[1] http://www.thejakartapost.com/news/2005/08/19/massive-blackout-hits-java-bali.html
[2] http://www.cnn.com/2008/US/02/26/florida.power/index.html

hardware and software from attackers [86]. Thus, ensuring the dependability of these sorts of applications takes on significance even beyond our initial work.

The absence of a test oracle makes these applications difficult to test, since the correct output cannot be known in advance. Further complicating the matter is the fact that software engineering practices are not traditionally prevalent in these domains (machine learning, scientific computing, simulation, etc.) [26]. Such software is often developed to explore something previously unknown in the domain of interest, making it difficult to identify requirements in advance. Typically the software developers have no formal knowledge of software engineering tools and techniques, and may be developing software on their own, acting as architect, developer, and tester [77]. Often, the developer may only test the application by checking that it runs to completion and does not produce an obvious error. Given the importance of such applications, and considering the general lack of software quality assurance in such domains, it is clear that a simple yet effective testing technique is of immediate importance.

2.2 Metamorphic Testing

Metamorphic testing has been suggested by Chen et al. as a way of testing applications that do not have test oracles [37] by ensuring that the software under test exhibits its expected "metamorphic properties".

A metamorphic property can be defined as the relationship by which the change to the output of a function can be predicted based on a transformation of the input [32]. Consider a function that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result: for instance, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the deviation from the mean would still be the same. Furthermore, other transformations will alter the output, but in a predictable way: if each value in the set were multiplied by 2,

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

Metamorphic properties can exist for an entire application, as well. Consider an application that reads a text file of test scores for students in a class, computes each student's average, and uses the function described above to calculate the standard deviation of the averages and determine the students' final grades based on a curve. The application itself has some metamorphic properties, too: permuting the order of the students in the input file should not affect the final grades; nor should multiplying all the scores by 10 (since the students are graded on a curve).

As a more complex example of how metamorphic testing can be used for applications in the domain of machine learning, anomaly-based network intrusion detection systems build up a model of "normal" behavior based on what has previously been observed. This model may be created, for instance, according to the byte distribution of incoming network payloads (since the byte distribution in worms, viruses, etc. may deviate from that of normal network traffic [186]). When a new payload arrives, its byte distribution is then compared to that model, and anything deemed anomalous causes an alert. For a particular input, it may not be possible to know *a priori* whether it should raise an alert, since that is entirely dependent on the model. However, if while the program is running we take the new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect has been found.

Clearly metamorphic testing can be very useful in the absence of an oracle: regardless of the values that are used in the test cases, if the relationships between the inputs and between their respective outputs are not as expected, then a defect must exist in the implementation.

That is, even if there is no test oracle to indicate whether $f(x)$ is correct, if the output $f(t(x))$ is not as expected, and the metamorphic property is sound, then the property is considered violated, and therefore a defect must exist.

2.3 Previous Work in Metamorphic Testing

Metamorphic testing was not originally devised as a solution to testing applications without test oracles. Rather, the concept was presented as a way of generating additional test cases from an existing set, particularly by suggesting that test cases that did not reveal defects did not necessarily have to be considered "failures", since they could be used to generate more test cases that may have fault-revealing power [32]. Metamorphic testing sought to extend the use of the algebraic properties of software [45, 103] to demonstrate that such properties could be used to create more test cases, even when starting from a limited number.

It was quickly noted that metamorphic testing could be applied to situations in which there is no test oracle [37, 202]. In some cases, these works looked at situations in which there cannot be an oracle for a particular application because the correct output cannot be known in advance [38]; in others, the work considered the case in which the oracle is simply absent or difficult to implement [28].

Most recently, Hu et al. conducted empirical studies to determine the effectiveness of metamorphic testing of applications without oracles [85], while others have applied metamorphic testing to specific domains of programs without test oracles such as bioinformatics [33], network simulation [35], and machine learning [195]. Other related work (in metamorphic testing or otherwise) is presented in Chapter 6.

This thesis contributes to the body of work on metamorphic testing by implementing improvements to the technique and measuring the improvements empirically. It also demonstrates that aspects of this improvement can be applied to testing applications in other domains that do, in fact, have test oracles.

2.4 Devising Metamorphic Properties

An open issue in the research on metamorphic testing is, "how does one know the metamorphic properties of the function or application?" Although others have looked into test case selection in metamorphic testing [34], i.e., choosing the test cases most likely to reveal defects, previous work assumes that the tester or developer will have sufficient knowledge of the system or function under test to identify its metamorphic properties, using application- or domain-specific properties.

To assist in the process of identifying metamorphic properties, in this section we describe some general guidelines that testers can follow and discuss how this fits into the overall software development lifecycle. As we are the first to present a classification of metamorphic properties, this is one of the major contributions of this thesis.

2.4.1 Mathematical Properties

Many programs without test oracles rely on mathematical functions (i.e., those that take numerical input and/or produce numerical output), since the point of such programs is to implement an algorithm and perform calculations, the results of which cannot be known in advance; if they could, the program would not be necessary. In Table 2.1, we categorize different classes of metamorphic properties that are common in mathematical functions. The classes are not meant to imply that the output will not be changed by such transformations, but rather that any change to the output would be predictable given the change to the input.

additive	Increase (or decrease) numerical values by a constant
multiplicative	Multiply numerical values by a constant
permutative	Permute the order of elements in a set
invertive	Take the inverse of each element in a set
inclusive	Add a new element to a set
exclusive	Remove an element from a set
compositional	Create a set from some number of smaller sets

Table 2.1: Classes of metamorphic properties

A simple example (for expository purposes only) of a function that exhibits these different classes of metamorphic properties is one that calculates the sum of a set of numbers. Consider such a function *Sum* that takes as input an array *A* consisting of *n* real numbers. Based on the different classes of metamorphic properties listed in Table 2.1, we can derive the following:

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

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

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

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

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

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

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

These are admittedly very trivial examples and do not fall under the category of programs without test oracles, but more complex numerical functions that operate on sets or matrices of numbers - such as sorting, calculating standard deviation or other statistics, determining distance in Euclidean space, etc. - tend to exhibit similar properties as well. Such functions are good candidates for metamorphic testing because they are essentially

mathematical, and demonstrate well-known properties such as distributivity and transitivity [133].

It is also the case that entire *applications* exhibit such properties, particularly in the domain of interest (in our case, machine learning) [131]. These applications and their metamorphic properties are discussed further in Section 2.5.

2.4.2 Considerations for General Properties

Although the classes of metamorphic properties listed in Table 2.1 can be useful in detecting defects, they are generally only applicable to functions and applications that deal with numerical inputs and outputs. Programs without test oracles tend to fall into this category (machine learning, scientific computing, optimization, etc.), but other programs in domains like computational linguistics and discrete event simulation work with non-numeric data, and these classes of properties may not be applicable.

As a general methodology for creating metamorphic properties, we propose the guidelines set out in the following four paragraphs. As a running example, we also provide possible metamorphic properties for applications from the domain of discrete event simulation. Such applications have no test oracle because the software is written to produce an output (the simulation) that was not already known in advance; if it *were* known in advance, then the simulator would not be necessary.

First, consider the metamorphic properties of **all applications in the given domain**. That is, there may be properties that are shared by all applications that operate in the domain, because of the nature of that domain. For instance, in discrete event simulation, regardless of the particular algorithm, there are generally "resources" that are modeled in the simulation. These resources may be doctors and nurses in a hospital, assembly line workers in an industrial factory, or postal workers who deliver mail. No matter what algorithm is used, and no matter what is being simulated, all of these share some common metamorphic properties. For instance, increasing the number of resources would be expected to lower

each resource's utilization rate, assuming the amount of work to be done remains constant. As another example, if the timing of all events in the simulation is multiplied by a constant factor, then the resource utilization should not change, since the ratio of the time spent working to the total time of the simulation would not be affected (because each are scaled up by the same factor).

Next, consider the properties of **the algorithm chosen to solve a particular problem in that domain**. That is, within the domain, one chooses an algorithm to solve a given problem, and that algorithm will itself have metamorphic properties. For instance, simulators can be used to model the process by which patients are treated in a hospital emergency room [60]. The process of simulating a patient's visit to the hospital emergency room might use an algorithm whereby steps and substeps are represented in a tree, and the entire process is essentially a traversal of that tree [156]. This architectural detail leads to numerous metamorphic properties relating to tree traversal: for instance, tree rotation is expected not to change the result of an inorder traversal; also, the tree can be broken into its constituent left and right subtrees, the combined traversal of which should be the same as the traversal of the entire tree. As another example, the selected algorithm may allow for steps of the process to run in parallel; a metamorphic property may be that changing the ordering of the parallel steps in the process specification should not affect the output (since they all execute at the same time, their ordering should not matter).

Then, consider the properties specific to **the implementation of the algorithm** used to solve the problem. A given application that uses the chosen algorithm may have particular metamorphic properties based on features of its implementation, the programming language it uses, how it processes input, how it displays its output, etc. For instance, in simulating the operation of a hospital emergency room, the process definition language Little-JIL [27] and its corresponding simulator tool (Juliette Simulator, or JSim [193]) may be used to specify the steps that an incoming patient goes through after arriving in the ER. In this implementation, the unique identifiers for the different resources (doctors, nurses, etc.) are

specified in a plain-text file; since this particular simulator treats all resources as being equal, this implementation exhibits the metamorphic property that permuting the order of the resources in the text file should not affect the simulated process.

Last, consider properties that are applicable only to **the given input** that is being used as part of the test case. It may be the case that some metamorphic properties of an application will only hold for certain inputs (this idea is explored further in the Future Work section in Chapter 7). Consider an input to the hospital emergency room simulation in which the number of resources is sufficiently large so that no patient ever needs to wait. For this particular input, increasing the number of resources should not affect the simulation, since those resources would go unused. But this particular property would not be expected to hold if there were too few resources, of course.

Although the examples provided here are specific to the domain of discrete event simulation (and simulation of a hospital emergency room in particular), such an approach could be used in other domains that have no test oracle, as well.

2.4.3 Automated Detection

We are not aware of any investigation into the automatic discovery of metamorphic properties, though this may be possible by building upon other techniques designed to detect similar characteristics of code. For example, dynamic approaches for discovering likely program invariants, such as Daikon [58] and DIDUCE [75], observe program execution and formulate hypotheses of invariants by relaxing different constraints on variables and/or using machine learning techniques to generate rules. Such techniques tend to focus on lower-level implementation details regarding application state and not on higher-level properties regarding function input and output, but could still be used as a basis for a dynamic approach.

The automatic detection of metamorphic properties may also build upon the work in the dynamic discovery of algebraic specifications [80], though that work has tended to focus

on data structures and abstract datatypes, and not on how an arbitrary function should react when its arguments are modified.

Of course, for any approach to automatically detecting code properties, a human "oracle" must decide whether the properties seem to be correct. If a defect prevents the detection of a metamorphic property, then the property will not be inferred and therefore no defect will be detected. On the other hand, a defect in the code might cause the detection of a property that the code should not, in fact, exhibit. Thus, as in invariant detection, the developer or tester must validate whether the suggested properties are sound [58].

It could be argued that static analysis techniques such as model checking or symbolic execution may be able to determine whether these properties hold, though such methods rely on an initial hypothesis of the property to be checked, and are not intended to discover the properties in the first place [42]. Furthermore, many metamorphic properties may be "hidden" within an implementation, and not detectable through analysis of the source code. As a very simple example, a sine function that uses a Taylor series $\sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!)$ obscures the metamorphic property that $\sin(x) = \sin(x + 2\pi)$.

Another approach would be to use machine learning techniques to automatically detect metamorphic properties by considering "similarities" in code. That is, if different pieces of code are known to exhibit a given property, then it may be possible to speculate that "similar" code (by some definition of "similarity") may exhibit the same property. This could also be done using techniques aimed at detecting code clones, which typically look for semantic and/or syntactic resemblance [65], but could conceivably be modified to indicate that two pieces of code exhibit the same metamorphic properties. Such approaches may be feasible in simple cases for the mathematical properties described in Table 2.1, though further investigation is required to determine how the approach fares on arbitrary pieces of more complex code.

The automated detection of metamorphic properties is outside the scope of this thesis, but is an important direction for potential future work.

2.4.4 Effect on the Development Process

Though there may be no "silver bullet" when it comes to devising the metamorphic properties of a given function or application, we would argue that in *any* software testing approach, the tester still must have some knowledge or understanding of the program in order to devise test cases. Semantic knowledge of the program or function to be tested is required for writing use cases, devising equivalence classes, creating test input, and designing regression tests [61, 135]. Even purely random testing approaches demand that the tester understand the input and output domain [73]. Thus, metamorphic testing is no different from other black-box techniques in that it is assumed that the tester will have enough knowledge of the code to create test cases (in this case, metamorphic properties), as guided by the program or function specifications and a general understanding of what the code is meant to do.

To facilitate this, we suggest that the metamorphic properties be identified as part of the planning and design phase, and included in the program specification. Thus, the application designer, who is likely to have the best understanding of how the program should react when its inputs are changed, can pass this knowledge on to the tester, who can then implement the specific test cases.

Metamorphic testing can also be applied to individual functions, and thus is suitable for use in unit testing and integration testing, particularly when there may be no oracle for the unit(s) being tested. Thus, in the implementation phase, developers can use metamorphic testing for subcomponents of the application; given that they are the ones who implemented the code, it should be easy for them to devise metamorphic properties, in the same way that it is easy for them to devise program invariants [43].

2.5 Applications in the Domain of Interest

In this section, we provide more details about the programs in the domain of interest (machine learning), and discuss how they exhibit the metamorphic properties listed above

in Table 2.1. These applications and properties will be used throughout the experiments in the following chapters. Other applications will be included in those experiments as well, and are described in the corresponding sections.

2.5.1 Machine learning fundamentals

In general, data sets used in machine learning (ML) consist of a collection of *examples*, each of which has a number of *attribute* values. The examples can be thought of as rows in a table, each of which represents one item from which to learn, and the attributes are the columns of the table.

In **supervised** ML, a *label* indicates how each example is categorized. In some cases the labels are binary: an example with a label of 1 is considered a *positive example*, and a 0 represents a *negative example*. In the motivating device failure application described in Section 2.1, though, the labels could be any non-negative integer, indicating how may times the device failed over a given period of time. Our investigation only considers binary labels, but the results are still generalizable. Figure 2.1 shows a small portion of a data set that could be used by supervised ML applications. The rows represent examples from which to learn, as comma-separated attribute values; the last number in each row is the label.

Supervised ML applications execute in two phases. The first phase (called the *training phase*) analyzes a set of *training data*; the result of this analysis is a *model* that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the *testing phase*), the model is applied to another, previously-unseen data set (the *testing data*) where the labels are unknown. In a classification algorithm, the system attempts to predict the label of each individual example; in a ranking algorithm, the output of this phase is a ranking such that, when the labels become known, it is intended that the highest valued labels are at or near the top of the ranking, with the lowest valued labels at or near the bottom.

Unsupervised ML applications also execute in training and testing phases, but in these

```
27,81,88,59,15,16,88,82,41,17,81,98,42, ..., 0
15,70,91,41, 5, 3,65,27,82,64,58,29,19, ..., 0
22,72,11,92,96,24,44,92,55,11,12,44,84, ..., 1
82, 3,51,47,73, 4, 1,99, 1,51,84, 1,41, ..., 0
57,77,33,86,89,77,61,76,96,98,99,21,62, ..., 1
...
```

Figure 2.1: Example of part of a data set used by supervised ML ranking algorithms

cases, the training data sets specifically do not have labels. Rather, an unsupervised ML application seeks to learn properties of the examples on its own, such as the numerical distribution of attribute values or how the attributes relate to each other. This model is then applied to testing data, to determine whether (or to what extent) the same properties exist. Anomaly-detection systems are types of applications that use unsupervised machine learning, as are data mining [74] and collaborative filtering [157].

2.5.2 MartiRank

Algorithm

MartiRank [70] was developed by researchers at Columbia University's Center for Computational Learning Systems as a ranking implementation of the Martingale Boosting algorithm [109], specifically with the electrical device failure application (described in Section 2.1) in mind.

In the training phase, MartiRank executes a number of "rounds". In each round, the set of training data is broken into sub-lists; there are N sub-lists in the Nth round, each containing $1/N$th of the total number of positive examples (i.e., examples with a label of 1). For each sub-list, MartiRank sorts that segment by each attribute, ascending and descending, and chooses the attribute that gives the best "quality". The quality is assessed using a variant of the Area Under the Curve [76] calculation that is adapted to ranking rather than binary classification. The model, then, describes for each round how to split the data set and on which attribute and direction to sort each segment for that round. In

the second (testing) phase, MartiRank applies the segmentation and sorting rules from the model to the testing data set to produce the ranking (the final sorted order).

```
1.0000,61,d
0.4000,32,a;1.0000,12,d
0.2500,18,d;0.5555,55,d;1.0000,41,d
```

Figure 2.2: Sample MartiRank model

Figure 2.2 shows a sample model created by MartiRank. In the first "round", shown on the first line, all of the examples are sorted by attribute 61 (indicated by the "61") in descending order (indicated by the "d"). In the second round, shown on the second line, the result of the first round is then segmented. The first segment contains 40% of the examples in the data set (indicated by the "0.4000") and sorts them on attribute 32, ascending. The rest of the data set is sorted on attribute 12, descending. The two segments are then concatenated to reform the data set, which is then segmented and sorted according to the next line of the model, and so on. A typical MartiRank model in the device failure application may have anywhere from four to ten rounds.

Metamorphic Properties

In determining the metamorphic properties of MartiRank, we first considered relationships that should not affect the output: either the model that is created as a result of the training phase, or the ranking that is produced at the end of the testing phase. For the training phase, if training data set input D produces model M, then we looked for input transformation functions T_i so that input $T_i(D)$ would also produce model M. Additionally, if testing data set input K and model L produce ranking $r(K, L) = R$, then we looked for input transformation functions T_i and model transformation functions T_m so that the combinations $r(T_i(K), L)$, $r(K, T_m(L))$ or $r(T_i(K), T_m(L))$ produce R as well.

Based on our analysis of the MartiRank algorithm, we noticed that it is not the actual values of the attributes that are important, but it is the *relative* values that determine the

model. Adding a constant value to every attribute, or multiplying each attribute by a positive constant value, should not affect the model because the model only concerns how the examples relate to each other, and not the particular values of the examples' attributes. The model declares which attributes to sort to get the best ordering of the labels; in Figure 2.1, if the values in any column were all increased by a constant, or multiplied by a positive constant, then the sorted order of the examples would still be the same, thus the model would not change. Additionally, applying a given model to two data sets, one of which has been created based on the other but with each attribute value increased by a constant, should generate the same ranking, based on the same line of reasoning. Thus, MartiRank exhibits metamorphic properties that we can classify as both **additive** and **multiplicative**: modifying the input data by addition or multiplication by a positive constant should not affect the output.

It should also be the case that changing the order of the examples should not affect the model (in the first phase) or the ranking (in the second). As MartiRank is based on sorting, in the cases where all the values for a given attribute are distinct, the sorted order will still be the same regardless of the original input order. Thus, MartiRank also has a **permutative** metamorphic property in that permuting the order of the input should not affect the output, albeit only limited to certain inputs.

We then considered metamorphic relationships that would affect the output, but in a predictable way. For the training phase, if training data set input D produces model M, then we looked for input transformation functions T_i so that input $T_i(D)$ would produce model M', where M' could be predicted based on M. Additionally, if testing data set input K and model L produce ranking $r(K, L) = R$, then we looked for input transformation functions T_i and model transformation functions T_m so that $r(T_i(K), L)$, $r(K, T_m(L))$ and $r(T_i(K), T_m(L))$ all can be predicted based on R. Keep in mind that in order to perform testing, we need to be able to have a predictable output based on R because we cannot know it in advance otherwise, since there is no test oracle.

We mentioned above that multiplying all attributes by a positive constant should not affect the model. On the other hand, multiplying by a negative constant clearly would have an effect, because sorting would now result in the *opposite* ordering. The effect on the MartiRank model, however, could easily be predicted, because the model not only specifies which attribute to sort on, but which direction (ascending or descending) as well. Consider that, if one were to sort a group of numbers in ascending order, then multiply them all by a negative constant, and sort in descending order, the original sorted order would be kept intact. In MartiRank, if in the original data set a particular attribute is deemed to be the best one to sort on, and a new data set is created by multiplying every attribute value by a negative constant, then that particular attribute will still be the best one to sort on, but in the opposite direction. The only change to the model will be the sorting direction. Thus, MartiRank displays an **invertive** metamorphic property, wherein it is possible to predict the output based on taking the "opposite" of the input. Like the permutative property, this property only holds in the case where all values are distinct, however, and would not be the case in situations in which there are repeating values and a stable sort is used.

This invertive property can also be seen in the testing phase. For data set input K, we define K' as its inverse, i.e., all attribute values multiplied by a negative constant. For model L, we define L' as its inverse, i.e., using the same attributes and partitioning but with the sorting directions all changed. We also define $R = r(K, L)$ as the ranking produced on data set K and model L, and R' as the inverse ranking, where the examples are ranked in "backwards" order. Based on the explanation above, we can expect that if $r(K, L) = R$, then $r(K', L')$ is also equal to R, because sorting the positive values ascending will yield the same ordering as sorting the negative values descending. It also follows, then, that $r(K', L)$ and $r(K, L')$ should both be equal to R', in which the ranking is the same but in the opposite direction.

Furthermore, once we know the model, it is easy to add an example to the set of testing data so that we can predict its final place in the ranking. Take, for example, the model

shown in Figure 2.2. In the first round, it sorts on attribute 61 in descending order; if we add an example to a testing data set such that the example has the greatest value in attribute 61, it will end up at the top of the sorted list. In the second round, the model sorts the top 40% (which would include our added example) on attribute 32 in ascending order; if we modify our added example so that it has the smallest value for attribute 32, it will stay at the top of the list. And so on. Knowing the model, we can thus construct an example, add it to the data set, and expect it to appear first in the ranking. We can thus say that MartiRank has an **inclusive** metamorphic property, meaning that a new element can be included in the input and the effect on the output is predictable. Similarly, MartiRank also shows an **exclusive** metamorphic property: if an example is excluded from the testing data, the resulting ranking should stay the same, but without that particular example, of course.

Last, the **compositional** property of MartiRank is simply demonstrated by duplicating all of the examples in the training data. That is, if data set input *D* produces model *M*, then a data set *D'* that consists of every example in *D* appearing twice should also produce the same model *M*, since the results of sorting the elements will stay the same, as will the percentage of positive examples used for creating the segments.

We note that all of these properties were validated with the MartiRank implementation used in the experiments in the following chapters.

2.5.3 Support Vector Machines

Algorithm

The Support Vector Machines (SVM) algorithm [183] belongs to the family of linear classifiers and is the foundation of numerous real-world applications [178]. In the learning phase, SVM treats each example from the training data as a vector of *N* dimensions (since it has *N* attributes), and attempts to segregate the examples with a hyperplane of *N*-1 dimensions. The type and shape of the hyperplane is determined by the SVM's "kernel": in our work, we only investigated the linear kernel, which is the default for the implementation

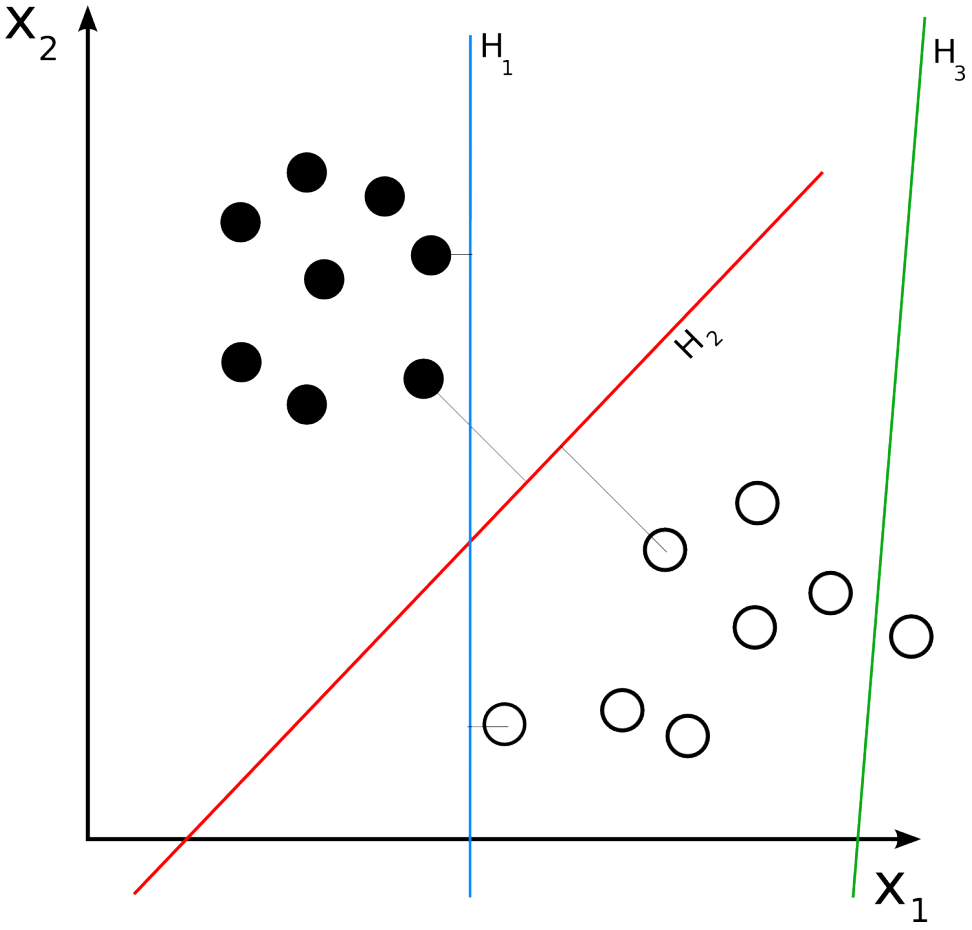

Figure 2.3: Data points separated by hyperplanes in SVM

we considered. The goal is to find the hyperplane with the maximum margin (distance) between the "support vectors", which are the examples that lie closest to the surface of the hyperplane; the resulting hyperplane is the model.

Figure 2.3 demonstrates a simple example.[3] Hyperplane H3 does not separate the two classes, indicated as black dots and white dots. H1 does separate the classes, but the support vectors (data points closest to the hyperplane) have a small "margin", or distance, from the hyperplane. Hyperplane H2 separates the classes with the maximum margin between the support vectors, and thus would be chosen as the model.

In the classification phase, examples in the testing data are classified according to which "side" of the hyperplane they fall on.

[3] http://en.wikipedia.org/wiki/Support_vector_machine

We tested the SVM implementation in the Weka 3.5.8 toolkit for machine learning [194], written in Java. While there are many other implementations of SVM, we chose the Weka implementation because it is open source and because of its popularity (over 1.6 million downloads on sourceforge.net as of November 2009, almost four times as many as the next most-downloaded open source machine learning toolkit).

Metamorphic Properties

As expected, SVM exhibits the same classes of metamorphic properties shown by MartiRank. Almost all of the transformations based on these metamorphic properties would be expected to result in a modification of the output compared to the original, but this modification should be predictable or easily be converted to the original output with additional transformations. For instance, if the training data set were transformed using an additive, multiplicative, and/or invertive relationship, then the corresponding model (hyperplane) should be affected by being shifted, expanded, or inverted in the N dimensions; if an example in the testing data set also had the same transformation(s) applied, the resulting classification when the new model is applied should be the same as that of the original model applied to the original data set, since the relation to the hyperplane should stay the same.

As with MartiRank, we would expect that SVM should exhibit the permutative property: it theoretically should produce the same model regardless of the order of the examples in the training data. In practice, though, this property does not always hold because of approximations that are used in the implementation [127]. An ML researcher familiar with SVM told us that because it is inefficient to run the 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 [168]. 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.

Thus, the models may demonstrate slight differences depending on which examples were in which chunks. We note, however, that although permuting the training data input does not always produce the exact same model, it generally produces a *semantically-equivalent* model, i.e., one that classifies examples in the testing data in the same manner as the original. Thus, for our purposes, we use this definition of the permutative property, and we validated this property with the Weka implementation used in our experiments.

Because every example in the training data influences the creation of the SVM model (hyperplane), even if in just a small way, it is clear that including an additional example or removing an example from the training data would have an unpredictable effect on the model; thus, it may appear that SVM does not have inclusive or exclusive metamorphic properties. However, classification algorithms in supervised machine learning *do* have properties related to including or excluding examples, but only once the classification of a particular example from the testing data is already known. For example, if a model M is created in the training phase and an example t is classified in the testing phase with label l, then we can duplicate all or some of the examples in the training data with label l, and the classification of t should remain the same, even if the model M does not; this is an inclusive metamorphic property if we simply insert a single new example, and a compositional property if we duplicate all examples with label l. Similarly, if we remove some of the examples in the training data that have labels other than l, again the classification of t would be expected to stay the same; this is an exclusive property. Both are a result of strengthening the relationship between the attributes in the training data and the label, and we have elsewhere argued that *all* classification algorithms would be expected to exhibit these particular properties [195].

2.5.4 C4.5

Algorithm

C4.5 [154] is a commonly used classification algorithm for building decision trees, in which branches represent decisions based on attribute values and leaves represent how the example is to be classified. Like other decision tree classifiers, it takes advantage of the fact that each attribute in the training data can be used to make a decision that splits the data into smaller subsets. In our experiments, we tested C4.5 release 8, implemented in C.

During the training phase, for each attribute, C4.5 measures how effective it is to split the data on a particular attribute value, and the attribute with the highest "information gain" (a measure of how well similar labels are grouped together [95]) is the one used to make the decision. The algorithm then continues recursively on the smaller sublists.

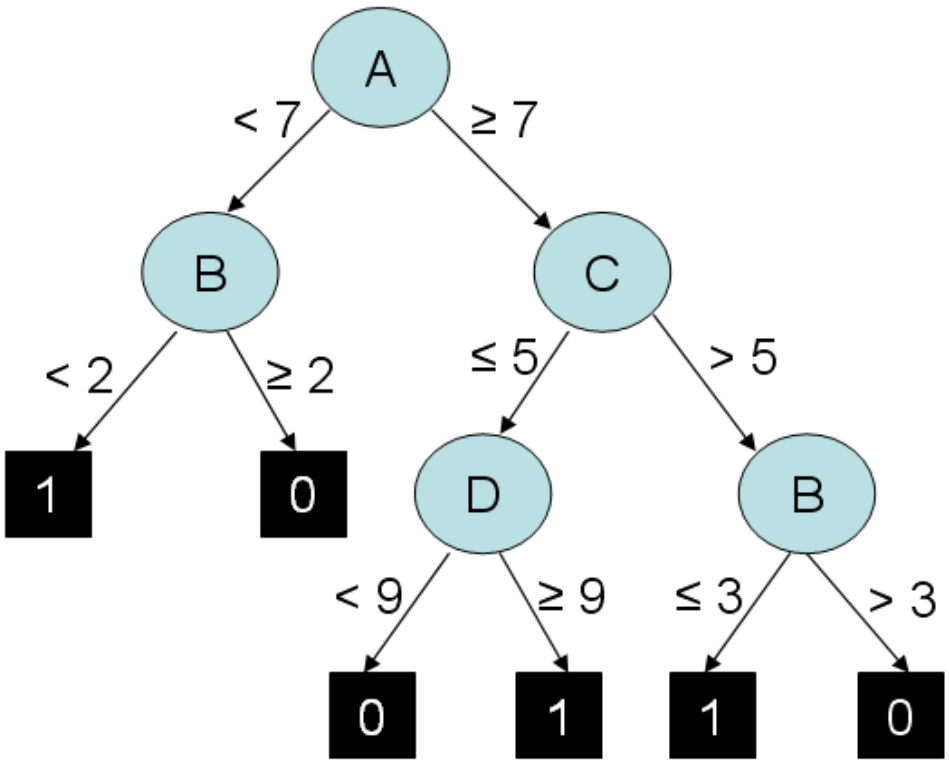

Figure 2.4: Decision tree used by C4.5

During classification, the rules of the tree are applied to each example, which is classified once it reaches a leaf. Figure 2.4 shows an example of a decision tree that could be used in the C4.5 algorithm (this example happens to be a binary decision tree, but that is not a limitation of C4.5). The circles represent nodes, at which different attribute values are evaluated. The edges coming from those nodes show the "direction" to traverse in the tree, based on the attribute values. Once a leaf is reached, the example can be classified, as the leaf specifies the label.

For instance, using the decision tree in Figure 2.4, if an example to be classified had the attributes {A=7, B=3, C=3, D=4}, then at the root of the decision tree, we would choose the right branch because A $\geq$ 7. At the next node, we look at the value of C, and choose the left branch because C $\leq$ 5. Then, we look at D, and choose the left branch again because D $<$ 9. Finally, we reach a leaf, and the example is classified with the label 0.

Metamorphic Properties

From looking at Figure 2.4, we can clearly see how C4.5 would exhibit a multiplicative metamorphic property: if the attributes in the training data were all multiplied by 10, for instance, the resulting tree would be expected to stay the same, except that the values on each edge should also be multiplied by 10. Then, if the attributes in the example used in the testing phase were also multiplied by 10, the classification should not change. The same case can be made for the additive property: as long as the same constant is added to the same attributes in both the training data and the testing data, the classifications should stay the same.

The same is true for the invertive property, but with a slight difference. For example, using the decision tree in Figure 2.4, if we apply the multiplicative property and multiply by 10, then the edges coming from the root node would change to "A $<$ 70" and "A $\geq$ 70". If we instead apply the additive property and add 10, then they would become "A $<$ 17" and "A $\geq$ 17". However, if we apply the invertive property and multiply by negative 1, in

order to get the same classification from the testing data (which would have to have the same transformation applied, of course), we must *reverse* the comparison signs. So the left edge would become "A ≥ -7", and the right would become "A < -7".

We noted, though, that in the C4.5 implementation we evaluated, the values on the edges were not *exactly* negated. This is due to a small constant ϵ that is used in the calculation of the information gain; because it is not negated during the metamorphic transformation, this affects the calculation. However, we observed the difference in values to be slightly less than 1%, which could be accomodated for by using a threshold for the comparison of the models, as discussed later in Section 3.3.

As with MartiRank and SVM, we would likewise expect that C4.5 should exhibit the permutative property, in that the model should not be affected by the order of the examples in the training data. And, as argued above and in [195], C4.5 has the same inclusive, exclusive, and compositional properties as all other classification algorithms, including SVM.

2.5.5 PAYL

Algorithm

MartiRank, SVM, and C4.5 represent three major categories of supervised machine learning: ranking, linear classification, and decision tree classification, respectively. To apply our approach to the category of *unsupervised* machine learning applications, we investigated PAYL [188], an anomaly-based network intrusion detection system (IDS) that was developed in Java by members of Columbia University's Intrusion Detection Systems Lab. Although PAYL can act as a standalone application, it has been incorporated within a commercial product that has been deployed in a number of corporate network environments [176].

Many intrusion detection systems are primarily signature-based detectors, and while these are effective at detecting known intrusion attempts and exploits, they fail to recognize

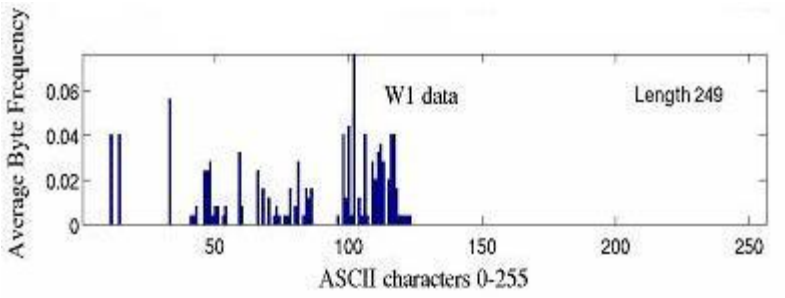

Figure 2.5: Sample payload byte distribution

new attacks and some variants of old exploits [4]. However, anomaly-based systems like PAYL are used to model normal or expected behavior in a system, and detect deviations that may indicate a security breach or an attempted attack. Anomaly-based IDS applications are considered unsupervised machine learning because there are no labels on the training data; rather, the system must learn on its own what is "normal" and what is "anomalous".

PAYL's training data simply consists of a set of TCP/IP network packets (streams of bytes), without any associated labels or classification. During its training phase, it computes the mean and variance of the byte value distribution for each payload length (the payload can be thought of as the "message" inside the network packet) in order to produce a model. Figure 2.5 shows an example of such a distribution [188]. During the second ("detection") phase, each incoming packet is scanned and the distribution of the byte values in its payload is computed. This new payload distribution is then compared against the model (for that payload length) using the Mahalanobis distance [114], which is a way of comparing two sets of data taking into account their correlations. If the distribution of the bytes in the new payload is above some configurable threshold of difference from the norm, PAYL flags it as anomalous and generates an alert.

[4] www.securityfocus.com/infocus/1663

Metamorphic Properties

Because the model generated by PAYL in the training phase represents the distribution of byte values in the TCP/IP payload (see Figure 2.5), it can be shown that it exhibits the additive property. Adding a constant value to each byte would simply shift the distribution, assuming overflows are handled by "wrapping" the values (e.g., 0xFF + 2 would become 0x01). Therefore, it would be easy to predict the effect on the model. Additionally, the categorization (as anomalous or not) of a packet in the testing phase would not change if it, too, had its bytes modified in the same manner.

As for a multiplicative property, though, it would not appear that multiplying the bytes in the payload by a constant would have a predictable effect on the output, particularly because the values need to be constrained to one byte, and there would be no one-to-one transformation based on multiplying that would meet that constraint. However, this is more of an issue of the particular application in this particular domain than it is of the approach itself, or of applying the approach to unsupervised machine learning algorithms.

Much of our analysis of PAYL focused on its permutative properties, primarily because some attackers may try to hide a worm or virus by permuting the order of the bytes, so as to trick a signature-based intrusion detection system. Of course, the model created by PAYL does not consider the order of the bytes, only their distribution, so a permutation should still result in the same model. At a higher level, because the model is created from a number of packets (not just a single one), permuting the order of the packets in the training data stream should also result in the same model.

PAYL also has an invertive property. An "inverse" of the distribution can be obtained by subtracting each byte value from the maximum (0xFF), so that frequently-seen values become less frequent, and vice-versa. If the same treatment is then applied to the payloads in the testing data as well, then the same alerts should be raised as before, since these values will still appear to be anomalous.

Aside from considering the distribution of byte values in creating its model, PAYL also

considers the existence (or absence) of payloads of certain lengths, and thus certainly has inclusive metamorphic properties. For instance, consider a model that generates an alert on a new payload because its length had never before been seen. If the particular payload were then included in the training data, it should no longer be considered anomalous. We would similarly expect PAYL to have exclusive metamorphic properties: if all payloads of a certain length were removed from the set of training data, then any messages of that length in the testing data would thus be considered anomalous because they had not previously been seen. The same holds true for port numbers, in addition to payload length.

2.6 Summary

In this chapter, we have discussed the metamorphic testing technique, presented guidelines for devising the properties of the software that are required for testing, and demonstrated that applications in machine learning exhibit such properties.

Next, we describe some of our enhancements to metamorphic testing, and show how they improve the state of the art in testing applications without test oracles.

Chapter 3

Automated Metamorphic System Testing

Although metamorphic testing is a simple technique in principle, clearly tool support is required in practice for all but the most trivial cases. The transformation of input data should be automated for large data sets, especially for input that is not in human-readable format. Similarly, comparing the outputs requires tools for cases in which the result sets are large, and/or if the outputs are not expected to be exactly the same.

It is certainly true that the testers can create one-off scripts to perform these transformations and comparisons, but to date there is no out-of-the-box testing framework that automates the way in which metamorphic testing is conducted in practice. In the same way that JUnit [89] provides a set of core functionality so that testers need not reinvent the wheel each time they perform unit testing, we intend to provide a new framework that automates the tasks performed by testers when they conduct metamorphic testing. Moreover, unlike JUnit, in which the testers still need to produce test code, the framework we seek to develop will allow them to merely specify the metamorphic properties using a simple notation, rather than needing to write any code.

The need for automation of the metamorphic testing process is particularly clear in the

cases in which slight variations in the outputs appear to indicate violations of metamorphic properties, but are not actually the result of errors in the code. This may come about due to imprecisions in floating point calculations that arise during additional executions of the program or function under test, and may lead to false positives (thinking that there is a defect when there actually is not). In such cases, the results need to be compared to within some threshold, or checked for semantic similarity.

Non-deterministic applications also present a challenge: to check whether a metamorphic property holds may require many executions of the code, and then some further analysis of the resulting outputs. Clearly these cases all would require tool support. To date, there has been very little work in investigating the application of metamorphic testing to non-deterministic applications, and yet many applications without test oracles may also rely on randomization, making them even more difficult to test.

In this chapter, we present an approach to automating metamorphic testing and describe an implementation of a testing framework called *Amsterdam* that facilitates the manner in which metamorphic testing is conducted in practice. To address the testing of non-deterministic applications, we introduce a new technique called *Heuristic Metamorphic Testing*. We also provide the results of new empirical studies conducted on real-world programs (both deterministic and non-deterministic) that have no test oracles to demonstrate the effectiveness of metamorphic testing techniques, and show that they are more effective than other common approaches.

The rest of this chapter is organized as follows: in Section 3.1, we further motivate the need for automation of the process by which metamorphic testing is conducted in practice. In Section 3.2, we describe the implementation of the Amsterdam framework for automating system-level metamorphic testing. In Section 3.3, we discuss the Heuristic Metamorphic Testing approach, and describe how it can be applied to non-deterministic applications. The results of our empirical studies are detailed in Section 3.4.

3.1 Motivation

Although it has been demonstrated that it is possible to use metamorphic testing to reveal previously-unknown defects in applications without test oracles [131, 195], the process by which the testing is conducted can benefit from automation, so that testers can be more productive by reusing existing testing frameworks and toolsets, and by running test cases in parallel whenever possible.

3.1.1 The Need for Automation

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

On a similar note, input data that is not human-readable (for instance, binary files representing network traffic) certainly require tools for transformation. For instance, as described in Section 2.5.5, the network intrusion detection system PAYL takes as input a collection of network packets represented as streams of bytes, the format of which must adhere to the strict TCP/IP conventions. Any transformation of this input would need to ensure that the new data set contains syntactically and semantically correct packets; however, it is quite difficult to know which parts of the byte stream to modify without tool support.

Additionally, the manual comparison of the program outputs can also cause problems. As with the input data, many applications for which metamorphic testing is appropriate produce large sets of output, and comparing them manually can be error-prone and tedious. In fact, Weyuker includes applications that produce a large amount of output in her definition of applications without test oracles [190].

Another issue involves metamorphic properties for which changes to the output are to be expected. For instance, in the C4.5 classifier discussed in Section 2.5.4, if the examples in the training data input have all attributes multiplied by two, then the decision tree should stay the same, but with the values at the nodes also multiplied by two. If the decision tree is large, it can be difficult to manually compare the two trees to make sure that they are as expected, and obviously a tool like "diff" is not sufficient because the trees are not expected to be exactly the same.

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

3.1.2 Imprecision

The problem with comparing program outputs - regardless of how it is done - is exacerbated by the fact that imprecisions in floating point operations in digital computers could cause outputs to appear to deviate, even if the calculation is actually correct programmatically. Although errors due to floating point imprecision are not the types of defects that we have set out to discover, comparing the outputs based on an expectation of equality may lead to a false positive, i.e., thinking that there is a defect when there actually is not [24].

Consider the simple case of the sine function, and the metamorphic property $\sin(\alpha) = \sin(\alpha + 2\pi)$. In practice, a defect-free implementation may cause a failure of the metamorphic test, due to imprecision in floating point calculations and the representation of

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

3.1.3 Non-Determinism

Non-deterministic programs may yield outputs that are *expected* to deviate between executions, and thus it may be impossible in practice to know whether the output is one that is predicted according to the metamorphic property, given the non-determinism. For instance, in the ranking algorithm MartiRank, described in Section 2.5.2, the result of the second (or "ranking") phase can be non-deterministic if there are missing attribute values in the data set: when the examples are sorted, those with unknown attribute values are randomly placed throughout the sorted list. If we permute the order of the examples in the input data, we would expect the output to stay the same if all attribute values were known. But if some are missing, the output will not stay *exactly* the same, because the placement of the examples with missing values is bound to change; this would also result in the placement of the known values being adjusted slightly. However, we may expect the new output to be "similar" to the original. The degree of expected similarity would need to either be specified by the tester or inferred automatically. In either case, tool support is required.

3.2 Automated Testing Framework

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

1. **Specify application's metamorphic properties.** The tester needs to specify the

application's metamorphic properties using a special syntax or scripting language. This specification should describe how to transform the input, and what the expected change to the output would be (Section 3.2.3).

2. **Configure framework.** The testing framework is configured so that it knows where to find the specification of the metamorphic properties, and how to run the program to be tested (Section 3.2.4).

3. **Conduct system testing.** The framework is invoked using test input data, as specified by the tester. The transformation of the input data, execution of the program, and comparison of the output are all automatically performed by the framework. The tester is notified about any violation of metamorphic properties, indicating that a defect has been found (Section 3.2.5).

Aside from facilitating metamorphic testing in the development environment, this technique can conceivably also be used to continually test the application as it runs in the deployment environment, as well. This must be done in such a manner that the end user only sees the results of the main (original) execution, and not from any of the others that are only for testing purposes; Section 3.2.6 investigates this further.

3.2.1 Model

Figure 3.1 shows the model of an implementation of such an automated approach. Metamorphic properties of the application are specified by the tester and then are applied to the program input. The original input is fed into the application, which is treated completely as a black box; a modified version of this input data is also produced, according to the metamorphic property. For each property to be checked, the corresponding input data is then fed into a separate invocation of the application, which executes in parallel but in a separate sandbox so that changes to files, screen output, etc. do not affect the other process(es). When the invocations finish, their results are compared according to the specification; if the

results are not as expected, a defect has been revealed. Although not reflected in Figure 3.1, it should be possible to execute more than two invocations of the program in parallel.

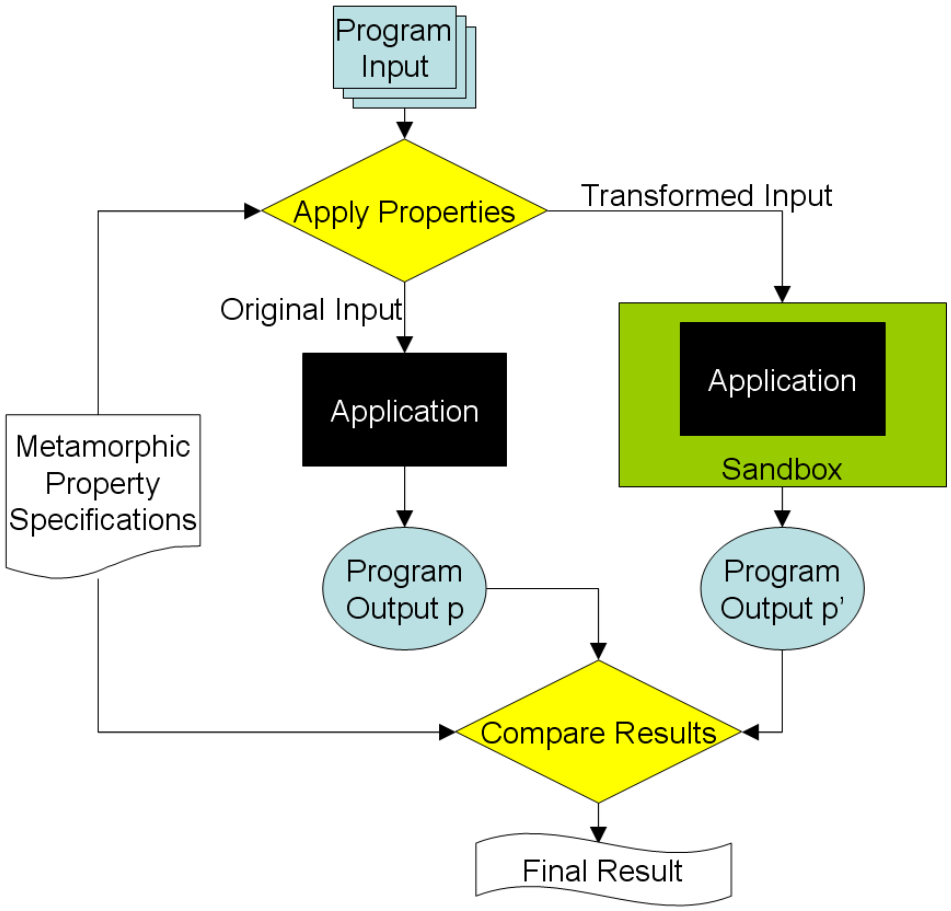

Figure 3.1: Model of Amsterdam testing framework

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

The rest of this section describes our implementation of a testing framework called *Amsterdam*, which enables an application to be treated as a black box so that system-level metamorphic testing can be automated without any modification to the code whatsoever.

As described above, multiple invocations of the application are run and their outputs are compared; however, the additional invocations must not affect the original process (or each other) and thus must run in a separate sandbox.

3.2.2 Assumptions

The current implementation of the Amsterdam framework assumes that: the program under test can be invoked from the command line; system input comes from files or from command line arguments; and, output is either written to a file or to standard out (the screen). This may limit the generality of this framework, but according to our preliminary investigations, these assumptions are typically not restrictive in applications in the domains of interest (particularly machine learning, but also discrete event simulation, optimization, etc.). Additionally, when input comes from database tables, mouse clicks, keystrokes, incoming network traffic, etc., an analogous unit testing approach can be used instead, so that metamorphic testing can be conducted at a more granular level; Chapter 4 describes such an approach.

3.2.3 Specifying Metamorphic Properties

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

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

For **input transformation**, the tester can describe how to modify the entire data set or only certain parts, such as a particular row or column in a table of data. As described in Section 2.4, we have identified seven general categories of metamorphic properties, and the framework supports out-of-the-box input modification functions to match each of these categories: adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing part of the data; adding additional data; and, composing a data set from smaller subsets.

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

For **output comparison**, the tester describes what the expected output should be in terms of the original output. In the simplest case, the outputs would be expected to be exactly the same. In other cases, the same transformations described above (adding, multiplying, etc.) for the input may need to be applied to the output before checking for equality. Additionally, the framework also supports checking for *inequality* if a change to the input is expected to cause a change to the output, even if that output cannot be precisely predicted. Last, if the output is non-deterministic, statistical or Heuristic Metamorphic Testing can be used, as described in Section 3.3.

Figure 3.2 demonstrates an example of a metamorphic property for system testing as specified in an XML file. On line 2, the name of the program to be run (in this case,

```
1  <TESTDESCRIPTOR>
2     <EXECUTION>/usr/bin/c-marti</EXECUTION>
3     <PARAMETERS>@input.training_data @output.model --no-permute --no-prob-dist</PARAMETERS>
4     <INPUT>
5        <VAR TYPE="csv_file" NAME="training_data" />
6     </INPUT>
7     <OUTPUT>
8        <VAR TYPE="text_file" NAME="model" />
9     </OUTPUT>
10    <POST_TEST>
11       <BRANCH NAME="main" />
12       <BRANCH NAME="test1">
13          @op_permute(@input.training_data)
14       </BRANCH>
15       <BRANCH NAME="test2">
16          @op_multiply(@input.training_data, 10)
17       </BRANCH>
18       <PROPERTY>
19          <ASSERT> @op_equal(@main.output.model, @test1.output.model) </ASSERT>
20          <ASSERT> @op_equal(@main.output.model, @test2.output.model) </ASSERT>
21       </PROPERTY>
22    </POST_TEST>
23 </TESTDESCRIPTOR>
```

Figure 3.2: Example of specification of metamorphic property for system-level testing

MartiRank) is specified, and on line 3 the names of the command-line parameters are given. For the purposes of metamorphic testing, the input is given the placeholder name "training_data" and the output is given the name "model"; other parameters specific to the execution of MartiRank are also specified on line 3. On lines 5 and 8, the types of files for the input and output are given, respectively.

The metamorphic properties to be tested are declared in lines 10-22. On line 11, we say that there is to be a "main" execution, the output of which will be shown to the user. On lines 12-14, we specify a test case called "test1" in which the input is to be permuted. On lines 15-17, we also specify another test case in which the elements of the training data should all be multiplied by 10. Lines 19 and 20 specify how to compare the outputs for the two metamorphic properties: in both cases, the new output (as a result of the test case) is expected to be the same as the original from the "main" execution.

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

If the framework does not support a specific transformation or comparison feature as required by the tester, functionality can be added by creating a separate component that

```
1  <TESTDESCRIPTOR>
2      <EXECUTION>/usr/bin/c-marti</EXECUTION>
3      <PARAMETERS>@input.training_data @output.model --no-permute --no-prob-dist</PARAMETERS>
4      <INPUT>
5          <VAR TYPE="csv_file" NAME="training_data" />
6      </INPUT>
7      <OUTPUT>
8          <VAR TYPE="text_file" NAME="model" />
9      </OUTPUT>
10     <FUNCTION NAME="switch" CLASS="MyOperators" METHOD="reverseDirection" TYPE="transform"/>
11     <POST_TEST>
12         <BRANCH NAME="main" />
13         <BRANCH NAME="test1">
14             @op_multiply(@input.training_data, -1)
15         </BRANCH>
16         <PROPERTY>
17             <ASSERT> @op_equal(@main.output.model, @op_switch(@test1.output.model)) </ASSERT>
18         </PROPERTY>
19     </POST_TEST>
20 </TESTDESCRIPTOR>
```

Figure 3.3: Example of specification of metamorphic property for system-level testing, with extended functionality

can be invoked by the framework, according to a specific programming interface (currently implemented in Java). This would also allow the tester to automate the transformation of other input formats not currently supported by the tool, or to compare other output formats.

For instance, Figure 3.3 shows the specification of another metamorphic property of MartiRank. In this one, multiplying all of the attributes in the training data by -1 (as specified on line 14) would not produce the exact same model, but rather the same model but with the sorting directions changed (see Section 2.5.2). The Amsterdam framework does not have a built-in function for comparing the outputs in such a manner, so the user needs to extend its functionality by creating a new component.

On line 10, the user introduces a new function called “switch”. This function name is mapped to a method called “reverseDirection” in a class called “MyOperators”, which the tester would have implemented. The type of the function is specified on line 10 as a “transform” function, meaning that the implementing method takes one argument, which is the data to be transformed. On line 17, when the outputs are to be compared, Amsterdam will invoke the method mapped to the name “switch”, which produces a new output that can then be checked against the original.

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

3.2.4 Configuration

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

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

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

The configuration also includes declaring what action to take if a metamorphic test fails. The tester can be notified that the test revealed unexpected behavior through an entry in

Amsterdam's execution log file and/or by a pop-up window.

3.2.5 Execution of Tests

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

The framework then starts additional invocations with the newly-generated inputs. When the test processes are to be run in parallel with the main process, the sandbox for each process is provided by simply creating copies of all the input files used by the test processes, and by redirecting screen output to a file so that the user does not see the results of the additional invocations of the program. To make the sandbox more robust, we have also integrated Amsterdam with a virtualization layer called a "pod" (PrOcess Domain) [147], which creates a virtual environment in which the process has its own view of the file system and process ID space and thus does not affect any other processes or any shared files. At this time, however, the framework sandbox does not include external entities such as the network or databases; this is discussed in Future Work (Section 7.2).

Once all processes are complete, the output files are then compared according to the specification of the metamorphic properties. If the output files are not as expected,

then a defect has been detected, and the appropriate action can be taken according to the configuration.

3.2.6 Testing in the Deployment Environment

The framework is also designed to be used in the production environment by the system's end users, so that metamorphic testing can continue even after deployment. Such "perpetual testing" [148] approaches have been shown to be effective at revealing defects that may not have been found before the software is deployed. This addresses one of the practical issues of metamorphic testing, which is "where does the test input come from?"

If the test invocations are run in parallel with the main one, the end user ideally would not even know that the testing was being conducted. In such cases, the results of the tests can be sent back to the development team for use in regression testing and program evolution [56]. The system administrator can specify the following configurations:

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

To date we have not investigated the mechanisms by which developers could actually use this information when defects are discovered, but we point out that others have previously looked into using similar field failure data to perform debugging and fault localization [44], or for regression testing [144]. We expect that such techniques are applicable in our case, as well, and leave this as future work.

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

3.2.7 Effect on Testing Time

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

3.3 Heuristic Metamorphic Testing

During the implementation of our Amsterdam framework, it became immediately apparent that false positives could be a problem if there were small deviations in the results of calculations that were expected to yield the same result. Additionally, many applications without test oracles rely on non-determinism, which limits the effectiveness of metamorphic

testing since it makes it more difficult to predict the expected outputs. To address this, we introduce a technique called *Heuristic Metamorphic Testing*, based on the concept of "heuristic test oracles" [83]. This variant of metamorphic testing permits slight differences in the outputs, in a meaningful way according to the application being tested. By setting thresholds and allowing for application-specific definitions of "similarity", we can reduce the number of false positives and address some cases of non-determinism. This new approach to conducting metamorphic testing is one of the major contributions of this thesis.

3.3.1 Reducing False Positives

In Heuristic Metamorphic Testing, output values that are "similar" are considered equal, where the definition of "similarity" is dependent on the application being tested. For instance, when the output is numeric (as in the sine function example in Section 3.1.2), a threshold can be set to check that the values are suitably close. As long as the difference between the values is below the threshold, then the outputs are considered sufficiently "similar", and no violation of the metamorphic property is reported.

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

For an example from the applications of interest, the C4.5 decision tree classification algorithm (discussed in Section 2.5.4) does not precisely adhere to the metamorphic property

based on negation, because a small constant ϵ that is used in determining the information gain affects the calculation. For almost all of the data sets used in the experiments below, the property seemed to be violated even in the "gold standard" implementation because of this small difference, i.e., the output was not exactly as expected. However, if we allowed for a tolerance of even just 1% when comparing the values, the property was never violated.

More complex cases may call for heuristics to check for semantic similarity. For example, in Section 2.5.3, we mentioned that the Support Vector Machines implementation did not strictly adhere to the metamorphic property based on permutation because of approximations used in the quadratic optimization algorithm. The values used in the specification of the model could conceivably be compared using a threshold, assuming the tester knows in advance how close the results should be, and assuming there is an easy way to compare the mathematical descriptions of the hyperplanes. Alternatively, we would expect the resulting model to be semantically similar to the original, in that the classification of examples in the testing data should be the same when each model is applied; the same is true in the case of C4.5 and the property based on negation, in fact. That is, even if the models are not exactly as expected, if they produce the same output in the classification phase, they can thus be considered semantically equivalent.

The Amsterdam framework supports techniques for considering such heuristics and either setting thresholds in the comparison of outputs, or specifying how to check semantic similarity, with the intent of reducing false positives. However, it has been argued that although the failure of a metamorphic test in such cases may not necessarily indicate a defect per se, it does reflect a deviation from expected behavior (albeit a very slight one), and thus it is useful to warn the user [195]. Therefore, Amsterdam can be configured to generate such a warning if so desired.

3.3.2 Addressing Non-Determinism

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

Statistical metamorphic testing

Guderlei and Mayer presented statistical metamorphic testing [71] as a solution to testing non-deterministic applications and functions that do not have test oracles. Consider a trivial example of a function r that takes two inputs x and y and returns a random integer in the range $[x, y]$. And, of course, $r(10x, 10y)$ would return a random integer in $[10x, 10y]$. However, we cannot simply specify the metamorphic property that $r(10x, 10y)$ should equal $10r(x, y)$, because of the randomization that occurs each time r is invoked.

Statistical metamorphic testing points out that, over many executions of $r(x, y)$, its output values will have a statistical mean μ and variance σ. Even if we cannot know whether μ and σ are correct, we would expect that many executions of $r(10x, 10y)$ should produce a mean 10μ and variance 10σ; of course, given a non-infinite number of executions, they may not be *exactly* 10μ and 10σ, but a statistical comparison (such as a Student T-test [67]) can be used. If the results are not statistically similar, then the metamorphic property has been violated.

Limitations of statistical metamorphic testing

Statistical metamorphic testing is limited to outputs that consist of a set of numbers whose statistical values, such as mean and variance, can be calculated. The approach is not necessarily applicable in the applications of interest presented here, for instance where the output could be a set in which the *ordering* is of prime concern, and the *values* of the set

elements are not important. For instance, the ranking application MartiRank may produce non-deterministic results in its ranking phase when there are missing values in the testing data set. Because the output is a listing of elements, there is no mean and variance to calculate, since the values themselves remain the same, but the ordering will be slightly different across multiple executions.

In Heuristic Metamorphic Testing, however, the results of the ranking algorithm can still be compared using some basic domain-specific metrics, such as the quality (measured using the Area Under the Curve [76], a common metric for comparing machine learning results) for each ranking, the number of differences between the rankings (elements ranked differently), the Manhattan distance (sum of the absolute values of the differences in the rankings), and the Euclidean distance (in N-dimensional space). Another metric is the normalized equivalence (or Spearman Footrule Distance), which explains how similar the rankings are (1 means exactly the same, 0 means completely in the opposite order) [173].

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

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

a statistical significance measure.

Examples

Consider an example from the MartiRank application described in Section 2.5.2. In its second ("ranking") phase, MartiRank applies the model to the examples in the testing data, sorting and segmenting them so that it yields the final ranking. If the data set contains any missing/unknown values, MartiRank will randomly place the missing elements throughout the sorted list, making the output non-deterministic. However, because the elements with known values should end up in approximately the same spot over multiple executions, we know that the final rankings should always be somewhat similar, and can use a heuristic (such as normalized equivalence) to measure that similarity. We also know that there is a metamorphic property that if, for instance, we multiply all values in the data set by 10, the final ranking should still be approximately similar over multiple executions.

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

As another example of Heuristic Metamorphic Testing of non-deterministic applications, consider the following case. If there are missing values in the training data for the first ("learning") phase of MartiRank, it will randomly place them throughout the sorted lists

when determining which attribute is best for sorting in that particular round (note that this is different from the previous example, which dealt with missing values in the *testing* data that is in the second, or ranking, phase). We know that there is a metamorphic property that if the values in the training data are all multiplied by 10, the model should be the same, but it will not be *exactly* the same because the randomness in dealing with missing values will cause multiple invocations of MartiRank to yield slightly different models.

The question then arises, "what heuristic can we use to compare the models?" One solution is to determine their similarity using semantic equivalence. That is, we can apply the two models to the same training data, and expect them to produce the same rankings. Yet, we know they won't be *exactly* the same, so we can use a heuristic such as normalized equivalence to measure the similarity of those outputs. But now the question arises "how similar should those rankings be?"

To address this problem, we can apply Heuristic Metamorphic Testing, by following these steps:

1. Run MartiRank N times on the original set of training data (with missing values) T_{train} to produce N models
2. Apply those N models to the set of testing data (with no missing values) T_{test} to produce N rankings
3. Build a profile P of those rankings according to a domain-specific heuristic, in this case by measuring their normalized equivalence
4. Create a new set of training data T'_{train}, which is the same as T_{train} but with all values multiplied by 10
5. Run MartiRank N times on the new set of training data T'_{train} to produce N models
6. Apply those N models to the set of testing data T_{test} to produce N rankings
7. Build a profile P'

8. Compare *P* and *P'*; if their difference is statistically significant, then the property is violated

In this case, we use both aspects of Heuristic Metamorphic Testing: running a non-deterministic application multiple times to determine its output profile; and using notions of semantic equivalence to compare outputs that are difficult to compare otherwise.

3.4 Empirical Studies

In their 2001 survey paper [15], Baresi and Young identified techniques for addressing the lack of a test oracle, including the use of runtime assertion checking, extrinsic interface contracts and algebraic specifications, formal specification languages, and trace and log file analysis. As far as we know, that paper represents the current state of the art in testing applications without test oracles, aside from the work on metamorphic testing. Others have more recently presented domain specific approaches for testing GUIs [120] or web applications [174], but we are considering the general class of programs that do not have test oracles.

To demonstrate the effectiveness of an automated metamorphic testing approach and determine how well it can detect defects in software without test oracles, we conducted three empirical studies on various real-world programs and compared the effectiveness of some of the proposed techniques. In the first study, we investigated the machine learning applications described in Section 2.5; in the second, we looked at applications without test oracles in other domains; and in the third study, we considered non-deterministic programs.

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

1. Is system-level metamorphic testing more effective than other techniques for detecting defects in applications without test oracles, particularly in the domains of interest?

2. Is Heuristic Metamorphic Testing more effective than other techniques for detecting defects in non-deterministic applications without test oracles?

3. Which metamorphic properties are most effective for revealing defects in the applications of interest?

Although others have conducted similar studies (most notably Hu et al. [85], discussed further in Related Work Section 6.2), the studies here are the first to consider applications in the particular domains of interest, and the first to evaluate the effectiveness of metamorphic testing when applied to non-deterministic programs. These studies are one of the major contributions of this thesis.

3.4.1 Techniques Investigated

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

Partial Oracle

As previously discussed, it is impossible to know whether the output of these programs is correct for *arbitrary* input, because there is no general test oracle to cover all cases. However, in some trivial cases, it may be possible to know what the correct output should be, based on analysis of the algorithm and understanding how it should perform under certain conditions. A "partial oracle" [151] is one that will indicate correctness or incorrectness of the output program that does not have a test oracle in the general case, but only for a limited set of test cases.

Determining the boundaries of such partial oracles (i.e., at what point does the program cease to have an oracle) is outside the scope of this work, but we assume for each of the applications in our studies that such oracles exist, either because the correct output can

easily be calculated by hand, or there is an agreed-upon correct output for the given input within the particular domain.

Weyuker points out that *"Experience tells us that it is frequently the 'complicated' cases that are most error-prone"* [190], thus limiting the effectiveness of the partial oracle. However, we assume that many developers of applications without test oracles would use such a technique, and as it is relatively simple and cheap to implement, we choose to include it in our studies.

Pseudo-Oracle

The concept of a pseudo-oracle is based upon that of "N-version programming" [31], in which independent programming teams develop an application (perhaps using different technologies or programming languages) from the same specification; then, identical sets of input data are processed and the results are compared. If the results are the same, that does not necessarily mean that they are correct (since the implementations may all have the same defect, for instance), but if the results are *not* the same, then a defect has likely been revealed in at least one of the implementations.

Although the effectiveness of N-version programming has been criticized from both a practical [2] and theoretical [93] point of view, the pseudo-oracle approach is still common in the machine learning community, for instance. Algorithms are typically prototyped in R or MATLAB before being implemented in C or Java; thus, we need to consider pseudo-oracles in our evaluation because of their widespread use in the domains of interest [172].

Assertion Checking

The use of assertions for fault detection has been employed since the early days of software development [43], and modern programming languages such as ANNA [111] and Eiffel [124], as well as C and Java, have built-in support for assertions that allow programmers

to check for properties at certain control points in the program. For instance, assertion checking may be used for program safety, such as to ensure that pointers are not null, array indices are in bounds, or that data structure integrity is maintained.

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

To aid in the creation of assertions, we used the Daikon invariant detection tool [58]. Daikon observes the execution of multiple program runs and creates a set of likely invariants, which can then be used as assertions for subsequent runs of the program. This has been shown to be effective at detecting defects like the ones we use in our studies [140]. Although it is possible to customize the types of invariants that Daikon can detect, in our experiments we only use its out-of-the-box features. Note that the invariants created by Daikon only include function pre- and post-conditions, and do not incorporate any assertions that are within the function itself; future work could consider the effectiveness of runtime assertion checking when using in-function invariants, though these would need to be generated by hand, as discussed in the Threats to Validity section below (Section 3.4.6).

Approaches Not Investigated

Formal specification languages like Z [1] or Alloy [87] could be used to declare the properties of the application, typically in advance of the implementation to communicate intended behavior to the developers. However, Baresi and Young point out that a challenge

of using specification languages as oracles is that "*effective procedures for evaluating the predicates or carrying out the computations they describe are not generally a concern in the design of these languages*" [15], that is, the language may not be suitable for describing how to know whether the implementation is meeting the specification. Although previous work has demonstrated that formal specification-based assertions can be effective in acting as test oracles [48], the specifications need to be complete in order to be of practical use in the general case [164]. Incomplete specifications can be used as partial oracles, though the extent to which this can be controlled in an experiment may introduce additional threats to validity. Thus, we do not consider formal specification languages in our evaluation.

Additionally, it may be possible to perform trace or log file analysis to determine whether or not the program is functioning correctly, if for instance it is conforming to certain properties (like a sequence of execution calls or a change in variable values) that are believed to be related to correct behavior; or, conversely, to see if it is *not* conforming to these properties. We have, in fact, investigated this technique previously [127], but noted that often the creation of an oracle to tell if the trace is correct can be just as difficult as creating an oracle to tell if the output is correct in the first place, assuming it is even possible at all. Therefore, we do not consider log file analysis in this study either.

3.4.2 Study #1: Machine Learning Applications

In this first study, we compare the effectiveness of metamorphic testing to that of three other approaches: partial oracles, runtime assertion checking, and pseudo-oracles. The test subjects chosen for this study come from the domain of supervised machine learning: C4.5, MartiRank, and SVM. The applications are described in more detail in Section 2.5.

Methodology

In this experiment, we used mutation testing to systematically insert defects into the source code and then determined whether or not the mutants could be killed (i.e., whether the

defects could be detected) using each approach. Mutation testing has been shown to be suitable for evaluation of effectiveness, as experiments comparing mutants to real faults have suggested that mutants are a good proxy for comparisons of testing techniques [3]. These mutations fell into three categories: (1) comparison operators were mutated, e.g., "less than" was switched to "greater than or equal"; (2) mathematical operators were mutated, e.g., addition was switched to subtraction; and (3) off-by-one errors were introduced for loop variables, array indices, and other calculations that required adjustment by one. Based on our discussions with the researchers who implemented MartiRank and PAYL, we chose these types of mutations because we felt that these represented the types of errors most likely to be made in these types of applications. All functions in the programs were candidates for the insertion of mutations; each variant that we created had exactly one mutation, i.e., we did not create any program variants with more than one mutation.

To determine which variants were suitable for testing, each was executed with the data sets described below. If the mutation yielded a compilation error, fatal runtime error (crash), an infinite loop, or an output that was clearly wrong (for instance, being nonsensical to someone familiar with the application, or simply being blank), that variant was discarded since any of the approaches would detect such defects. We also ensured that each mutant was on the execution path of at least one of the data sets used as test cases.

In the SVM implementation, we ended up with a total of 85 separate versions that were suitable for our experiment, each with exactly one mutation. For C4.5, we had 28, and for MartiRank, 69. The variation in the number of mutants for each application is due to the different sizes of the source code, the different number of points in which mutations could be inserted, and the different number of mutants that led to fatal errors or obviously wrong output. The breakdown of mutants by type is specified in Table 3.1.

Creating Partial Oracles. As the SVM and C4.5 algorithms are fairly commonly used in the machine learning community, there exist sample data sets for which the correct output is actually known; these are used by developers of SVM and C4.5 implementations to make

Mutant Type	C4.5	MartiRank	SVM
Comparison	8	20	30
Math	15	23	24
Off-by-one	5	26	31
Total	**28**	**69**	**85**

Table 3.1: Types of mutants used in Study #1.

sure there are no obvious defects in their code. Of course, producing the correct output for this input does not ensure that the implementation is error-free, but these data sets can be used as partial oracles. We used the "iris" and "golf" data sets from the UC-Irvine machine learning repository [138] for both SVM and C4.5, since the data sets are relatively small in size and the correct output can be calculated by hand. For both of those data sets, we created four variants of each, using different values or different ordering of examples, so that the output would still be easy to calculate, for a total of eight data sets.

For MartiRank, we hand-crafted two data sets for which we could know in advance what the expected output should be, and confirmed our expectation using the "gold standard" implementation, which we assumed to be error-free (this assumption is discussed in Section 3.4.6 below). As with the SVM and C4.5 data sets, we created four variants of each one, each of which would yield an output that could be known.

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

Pseudo-Oracles. We were fortunate that the three machine learning algorithms selected for this experiment all had multiple implementations that were readily available to us.

For SVM, we selected LIBSVM [29], which is implemented in C, as the pseudo-oracle for Weka's SVM implementation in Java. For C4.5, we chose the J48 implementation in Weka 3.5.8 as the pseudo-oracle for the C implementation. And for MartiRank, we selected

the Perl implementation, which was the original prototype of the algorithm, to act as the pseudo-oracle for the C version.

Metamorphic Testing. For the three applications, we devised metamorphic properties according to the guidelines described in Section 2.4. Each property was verified with the "gold standard" version to make sure that the property was, in fact, expected to hold for the data sets described below. The properties are defined as follows:

1. Permuting the order of the examples in the training data should create a semantically-equivalent model, i.e., one that yields the same classification (or ranking, in the case of MartiRank) of examples in the testing data.

2. Multiplying each attribute value in the training data by a positive constant (in our case, ten) should create a semantically-equivalent model.

3. Adding a positive constant (in our case, ten) to each attribute value in the training data should create a semantically-equivalent model.

4. Negating each attribute value in the training data, followed by negating each attribute value in the testing data, should result in the same classification (or ranking) for each example.

For SVM and C4.5, we performed the experiment using six data sets from the UC-Irvine machine learning repository [138], listed in Table 3.2. The sets are all relatively small in size because the SVM algorithm can take over an hour to run on larger data sets [133].

These data sets could not be used for MartiRank because MartiRank requires numeric labels, so we used 10 randomly-generated data sets that covered a variety of equivalence classes (using a "parameterized random testing" technique [129]), to try to obtain line coverage similar to the data sets used in the partial oracle experiment. These data sets are listed in Table 3.3.

For each data set, each mutated version was executed with both the original and the data set modified according to each metamorphic property, and the results were compared:

Name	# examples	# attributes
golf	14	4
iris	150	4
wine	178	13
hepatitis	155	19
heart	303	75
glass	214	10

Table 3.2: Data sets used for C4.5 and SVM with metamorphic testing and runtime assertion checking.

Name	# examples	# attributes
data1	10000	100
data2	20000	100
data3	20000	100
data4	10000	100
data5	1000	40
data6	1000	200
data7	1000	100
data8	1000	100
data9	100	200
data10	10000	200

Table 3.3: Data sets used for MartiRank with metamorphic testing and runtime assertion checking.

if the outputs were not as expected, then the mutant was considered to be "killed" (i.e., the defect was found). Note that, in this experiment, all three applications were deterministic; we did, however, need to use the Heuristic Metamorphic Testing features of the Amsterdam framework because we were looking for semantic equivalence in the models in some cases.

Invariant Detection. To create the set of invariants that we could use for runtime assertion checking, we applied Daikon to the "gold standard" (i.e., with no mutations) of the three applications. We executed the applications multiple times, with the different data sets described in Tables 3.2 and 3.3. This was done in order to reduce the number of "spurious invariants" that Daikon might produce. For instance, if a program is only run once with an input called "input.data", Daikon will detect that the variable representing the input file is always equal to "input.data" and create an invariant as such; clearly this is spurious

because it only happens to be true in this one particular case.

Once the set of invariants had been created, we executed each mutated version with the same data sets used for metamorphic testing (six for SVM and C4.5, ten for MartiRank) and used Daikon's invariant checker tool to ensure that none of the invariants were being violated. If a violation occurred, then the defect had been found.

Note that in practice, it may not be possible to derive a complete set of invariants from the "gold standard": after all, if one knew that the implementation were the gold standard, testing it would not be necessary. However, in this experiment, we wanted to create a set of invariants that would best reflect the correctness of the implementation, and this seemed the best way to automate that. An alternative would have been to hand-generate a set of program invariants, but this would have required more detailed understanding of the implementation details, and we did not have access to the developers of the C4.5 and SVM implementations. Rather, we aimed to compare the techniques assuming that the tester only had an understanding of what the applications are meant to do, and not how they are implemented.

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

The complete set of Daikon-generated invariants is too lengthy to be enumerated here, and is instead listed in Appendix A of the tech report [128] describing this study.

Results

Each of the four techniques described above (partial oracle, pseudo-oracle, metamorphic testing, and runtime assertion checking) was applied to all mutated variants of the three applications investigated in this experiment. The goal was to determine what percentage of

the mutants were killed by each approach.

Table 3.4 shows the effectiveness (percentage of distinct defects found) for the partial oracle, metamorphic testing, and assertion checking. We have omitted the results of testing with pseudo-oracles for reasons that are described at the end of the Discussion section.

Overall, metamorphic testing was most effective, killing 97.8% of the mutants in the three applications. Assertion checking found 84% of the 182 defects, and the partial oracle detected 80.7%.

App.	**# Mutants**	**Partial Oracle**	**Metamorphic Testing**	**Assertion Checking**
C4.5	28	22 (78.5%)	26 (92.8%)	28 (100%)
MartiRank	69	47 (68.1%)	69 (100%)	57 (82.6%)
SVM	85	78 (91.7%)	83 (97.6%)	68 (80.0%)
Total	**182**	**147 (80.7%)**	**178 (97.8%)**	**153 (84.0%)**

Table 3.4: Distinct defects detected in Study #1.

To further assess the results, Table 3.5 shows the number of mutants killed by none of the approaches, only one of the approaches, two of the three approaches, and all three. This gives an indication of where the overlaps are between the different testing techniques.

	Number of Mutants
Not killed	0
Partial Oracle only	1
Metamorphic Testing only	9
Assertion Checking only	2
Partial Oracle and Metamorphic Testing only	19
Partial Oracle and Assertion Checking only	1
Assertion Checking and Metamorphic Testing only	24
All three	126
Total	**182**

Table 3.5: Breakdown of defects detected in Study #1.

This table demonstrates that metamorphic testing was able to detect nine defects not

found by the other approaches, whereas there were only four defects that it could not find. By way of comparison, the partial oracle only found one defect not found by the others, but missed 35; and assertion checking detected two not found by the others, but missed 29. It is clear that by these various metrics, metamorphic testing is the most effective overall.

Discussion

Here we discuss the results of the empirical study and evaluate why certain testing approaches are more effective than others.

C4.5. As opposed to the other two applications, runtime assertion checking was more effective than metamorphic testing at revealing the C4.5 defects, although only slightly so (28 mutants killed, compared to 26). As shown below in Table 3.7, the mutants not killed by metamorphic testing were both math mutants, in which a calculation was modified. In both cases, this resulted in a violation of an assertion, but not a violation of a metamorphic property.

One of the two defects is found in the code in Figure 3.4, which comes from the "FormTree" function used to create nodes of the decision tree. Lines 179-182 use a for-loop macro to initialize the values of an array `ClassFreq`, which is used to calculate the frequency with which a class (or label) appears in the training data. Line 183 then uses the macro to iterate values of `i` from `Fp` (the first index of the array) to `Lp` (the last index). On line 185, the `Class` function is used to determine the class of the item from the training data, and the `ClassFreq` array is updated with the corresponding weight.

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

to be violated.

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

Figure 3.4: Sample code in C4.5 implementation. A defect on line 185 causes a violation of the assertion checking, but not of the metamorphic property.

We see from Table 3.1 that C4.5 had a proportionally higher number of math mutants, compared to SVM and MartiRank; as the example in Figure 3.4 is a math mutant, as well, it may seem that assertion checking is more effective for these types of defects. However, we observed in all three applications that metamorphic testing is approximately as effective for all three types of defects (comparison operator mutations, math operator mutations, and off-by-one errors), and we attribute this result perhaps to the relatively small sample of usable mutants that we were able to insert in the C4.5 code.

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

Although this seems to put the partial oracle at a disadvantage, we point out that both the partial oracle data sets and the data sets used for metamorphic testing were attaining

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

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

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

Note that the definition of a "non-testable program" is not only limited to software for which an oracle does not exist: it can also include software for which the creation of an oracle is too difficult [190], and the partial oracle by its nature tends to address a smaller, simpler part of the input domain. This does serve to point out, though, that the metamorphic testing and runtime assertion checking techniques can be used with arbitrary input data of any size, thus improving their effectiveness.

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

Figure 3.5 shows a sample of the SVM code that was tested in this experiment. The array `m_class` identifies the classification, or "label", for each example in the training data. This particular piece of code attempts to segregate the examples into those that have a label of 1 (into set `m_I1`), and those that do not (into set `m_I4`). An off-by-one mutant has been inserted into line 524, such that the for loop omits the final element of `m_class`.

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

Figure 3.5: Snippet of code from Weka implementation of SVM, with an off-by-one error in the for-loop condition on line 524

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

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

Assertion checking fails to find this defect because the relevant invariants simply deal with sizes of the sets (e.g., "`m_I1.size` > 0") and the fact that `m_class` does not change. Despite the mutant in line 524, these invariants still hold. This is not to say that the invariants are wrong or are even incomplete; rather, the use of invariants alone is not sufficient to reveal this defect.

Pseudo-Oracles. Strictly speaking, the pseudo-oracles for the three applications killed all but one of the mutants, i.e., the output of the pseudo-oracle differed from that of the mutated version in almost every case. However, none of the pseudo-oracles that we employed were actually suitable for our purposes, because none of them consistently produced the exact same output as the gold standard versions that we used in the experiment (i.e., the ones without the mutants). For C4.5, there were only very minor differences that appeared at lower levels of the decision tree in the Weka J48 implementation, and this was only for one of the six data sets; thus, one could argue that this pseudo-oracle was "good enough".

However, for both MartiRank and SVM, the other implementations we selected produced different results for *all* of the data sets. This essentially means that the pseudo-oracles we chose for those applications have a 100% false positive rate: they supposedly revealed defects (because the outputs were different), even though no defect existed. Upon further investigation, we realized that this is partly by design: for SVM, the Weka implementation used in the experiment performs different types of optimizations when calculating the hyperplane so as to improve performance at the cost of accuracy. However, for MartiRank, this difference was a result in different interpretations of the specification on the part of the developers [127]. These differences mean that not only are the outputs (the models) of the implementations slightly different, but they are semantically different as well: applying the different models to the same training data set also produced differing results in the classification/ranking phase. Therefore, in addition to the other criticisms of N-version programming, we can also point out that care must be chosen in selecting alternative imple-

Application	# Mutants	Permute	Multiply	Add	Negate	Total
C4.5	28	13	3	4	23	26 (93%)
MartiRank	69	69	1	1	38	69 (72%)
SVM	85	76	23	25	56	83 (97%)
Total	**182**	**158**	**27**	**30**	**117**	**178 (98%)**

Table 3.6: Summary of results of mutation testing using metamorphic testing

mentations of an algorithm as pseudo-oracles, as there may be deliberate (or accidental) differences that lead to false positives.

Additionally, in one case, the pseudo-oracle for SVM had a false negative: the mutated version coincidentally produced the same output as the pseudo-oracle, so it appeared that no defect existed. Even though this was very rare (it only happened once), this also adds to the argument that the pseudo-oracle approach is not without its flaws.

Analysis of Metamorphic Testing Results

The results of metamorphic testing of all three applications are summarized in Table 3.6. For each application, the second column shows the total number of mutants that were suitable for use in the testing (i.e., that did not produce an obvious error). The next four columns show how many of those mutants were killed by metamorphic properties based on permuting the input, multiplying the input values by a constant, adding a constant to each input value, and negating each input value, respectively. The last column shows the total number of distinct mutants killed by metamorphic testing (a mutant may be killed by multiple different metamorphic properties), and the overall percentage.

Following, we analyze the results for each of the three applications.

C4.5 Table 3.7 lists, for each of the three types of mutations inserted into C4.5, the number of mutants that were killed by metamorphic testing with the different types of properties as listed above.

The metamorphic property based on negating the input proved to be the most reliable means of killing the C4.5 mutants (23 of 28, or 82%), and was more or less equally effective

Mutation	# Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	8	8	0	1	8	8 (100%)
Math operators	15	2	3	3	11	13 (86%)
Off-by-one	5	3	0	0	4	5 (100%)
Total	**28**	**13**	**3**	**4**	**23**	**26 (93%)**

Table 3.7: Results of mutation testing for C4.5 using metamorphic testing

for all three types of mutations. This is because the nodes of the decision tree contain clauses such as "if $attr_n > \alpha$ then class = C", where $attr_n$ is some attribute, α is some value, and C is the classification. If all the training data were negated, then the clause is expected to become "if $attr_n \leq -\alpha$ then class = C", which requires both the comparison operator and the sign of α to be switched. However, in most of the cases, only one or the other was switched, so that in the classification phase, elements in the testing data were not correctly classified. Because C4.5 also involves calculations (to determine which splitting of data provides the best information gain), other mutations caused the value of α to be changed when the training data values were negated.

Table 3.8 shows the effectiveness at revealing defects for each of the data sets used in the C4.5 experiment. These data sets are from the UC-Irvine machine learning repository [138] and are summarized above in Table 3.2. All of them are similarly effective, except for the "golf" data set, which only killed three of the mutants. This particular data set is much smaller than the others, having only 14 examples, compared to the others, which had at least 150 examples. Because of the small size, the generated decision tree is quite small, and thus there are fewer points for the metamorphic properties to detect defects.

Mutation	# Mutants	iris	golf	heart	glass	wine	hepatitis
Comparison	8	8	3	7	6	7	8
Math	15	8	0	7	7	4	10
Off-by-one	5	3	0	4	5	3	1
TOTAL	**28**	**19**	**3**	**18**	**18**	**14**	**18**

Table 3.8: Number of mutants killed per data set for C4.5

MartiRank. Metamorphic testing killed all 69 mutants in the MartiRank application. As shown in Table 3.9, the property based on permuting was particularly effective, with the negation property less so, and the properties based on multiplying and adding not at all.

Mutation	# Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	20	20	1	1	16	20 (100%)
Math operators	23	23	0	0	12	23 (100%)
Off-by-one	26	20	0	0	10	26 (100%)
Total	**69**	**69**	**1**	**1**	**38**	**69 (100%)**

Table 3.9: Results of mutation testing for MartiRank using metamorphic testing

Consider a simple function to perform bubble sort (although MartiRank uses merge sort, not bubble sort, the bubble sort example is easier to understand). One would expect that permuting the input should not affect the result, as the sorted order does not depend on the original order. However, if there were an error in the implementation, then permuting the order of the inputs may yield a different result, revealing the defect.

In the erroneous implementation in Figure 3.6, the condition `i < A.length - 1` on line 2 should simply be `i < A.length`. Here, inputs like A = {3, 1, 2, 5, 4} or A = {4, 3, 1, 2, 5} and many others would correctly return the sorted array. However, if we permute the input and happen to set A = {2, 3, 5, 4, 1} (or any other input in which the smallest value is at the end), it would incorrectly return an unsorted list, and metamorphic testing reveals the defect.

On the other hand, a metamorphic property based on multiplication or addition would not find this defect. For instance, A = {2, 3, 5, 4, 1} would return {2, 3, 4, 5, 1}, and A = {20, 30, 50, 40, 10} would return {20, 30, 40, 50, 10}, as expected according to the metamorphic property. Thus, changing the values in A does not reveal these types of defects, but changing their order does.

Table 3.10 shows the sensitivity of the results to the different data sets used in the

```
1 bubble_sort(A) {
2   for (i = 0; i < A.length - 1; i++)
3     for (j = 0; j < A.length - i; j++)
4       if (A[j] > A[j+1])
5         swap A[j] and A[j+i]
6   return A;
7 }
```

Figure 3.6: Mutated function to perform bubble sort with off-by-one error on line 2

experiment for MartiRank. Almost all of the data sets were equally effective at revealing defects, except for data set #6, which killed all but one of the 69 mutants. One of the reasons why it may have been so effective is that this data set had 200 attributes, whereas most of the others had no more than 100. This means that there would be twice as many opportunities for errors to arise in the sorting of attributes and the calculation of the quality of the results. Thus, it follows that more of the defects would be revealed.

	Comparison	Math	Off-by-one	**TOTAL**
COUNT	**20**	**23**	**26**	**69**
data1	18	15	17	**50**
data2	20	15	17	**52**
data3	18	14	18	**50**
data4	19	15	16	**50**
data5	19	14	19	**49**
data6	19	23	26	**68**
data7	18	15	15	**48**
data8	18	14	15	**47**
data9	13	14	15	**42**
data10	19	14	16	**49**

Table 3.10: Number of mutants killed per data set for MartiRank

SVM. For SVM, permuting the input was the most powerful metamorphic property in terms of revealing defects: 76 of the 85 mutants (89%) were killed, as shown in Table 3.11.

Permuting the input was particularly effective in killing the off-by-one mutants in SVM (30 out of 31, or 97%). In these mutations, for-loops omitted either the first or last value in an array, thus the mathematical calculations would yield different results because different

Mutation	# Mutants	Permute	Multiply	Add	Negate	Total
Comparison operators	30	23	9	2	19	29 (96%)
Math operators	24	23	3	12	19	23 (95%)
Off-by-one	31	30	11	11	18	31 (100%)
Total	**85**	**76**	**23**	**25**	**56**	**83 (97%)**

Table 3.11: Results of mutation testing for SVM using metamorphic testing

permutations meant that different elements were being left out. For instance, consider a function $f(A) = \sum_i A_i$, where A is an array of values. One would expect that permuting the order of the elements in A would not affect the result. But clearly if, say, the first element of A is not included in the sum, then permuting the elements will put a different one first, and thus the result will change, in violation of the metamorphic property.

Table 3.12 shows the number of mutants killed for each data set used in the SVM experiment. The numbers are fairly consistent except that the "iris" data set was very effective for the off-by-one mutants, but not as effective for the comparison mutants.

Mutation	# Mutants	iris	hepatitis	heart	glass	wine	golf
Comparison	30	17	28	27	26	25	24
Math	24	18	19	23	21	21	20
Off-by-one	31	31	18	18	26	13	20
TOTAL	**85**	**66**	**65**	**68**	**73**	**59**	**64**

Table 3.12: Number of mutants killed per data set for SVM

Additional Results

As part of our study, we also investigated the anomaly-based intrusion detection system PAYL, described in Section 2.5. In our experiments with PAYL, we created 40 mutants for testing, but only two were killed using our approach. One reason for this poor performance is that we were only able to automate two metamorphic properties for PAYL: permuting the order of the packets in the training data set; and permuting the order of the bytes within the

payload ("message") in each packet. Although permutation of the input proved to be an effective technique for detecting defects in SVM and MartiRank, PAYL is purely concerned with distribution of values and not at all with their ordering or relationship to each other; thus, it follows that permuting the input would have very little effectiveness at killing the types of mutants that we introduced.

By way of comparison, runtime assertion checking using Daikon-generated invariants killed 13 of the 40 mutants. This is not a great result either, but we mention this here to demonstrate that metamorphic testing is not a silver bullet for applications that have no test oracle, and that its effectiveness relies heavily on a combination of the types of metamorphic properties, the types of defects being targeted, and the nature of the application itself.

3.4.3 Study #2: Applications in Other Domains

In the next study, we sought to apply the different techniques to applications in other domains, including discrete event simulation, information retrieval, and optimization.

Test Subjects

Discrete event simulators can be used to model real-world processes in which events occur at a particular time and affect the state of the system [152]. Simulators can be considered to have no test oracle because if the results of the simulation could be known in advance, the software would not be necessary.

In this study, we tested the simulator JSim [193], which is implemented in Java and has been used in the modeling of the flow of patients through a hospital emergency room [156]. The input to JSim consists of a *process*, represented as a tree of steps and constituent substeps; a set of *agents* who perform the steps, such as patients in the emergency room; a set of *resources*, which may indicate things like how many doctors, nurses, hospital beds, etc. are available to be used in the steps; and an *oracle*, which dictates when process events should occur and how long they should take (not to be confused with a "test oracle", of

course).

When the simulation is run, each agent is assigned an *agenda*, or a list of steps to perform. The simulator goes through the process, and the oracle indicates whether it is time for each agent to perform the next step on its agenda. If it is, and sufficient resources are available, then the agent performs the step and moves on to the next one on its agenda; if no resources are available, then depending on the configuration, the agent either blocks (waits for the required resource) or throws an exception, in which case the process is terminated. This continues until all agents have completed their agendas. The output of the simulation is the sequence of events (steps) that took place, the times at which they started and ended, and the agents and resources involved.

The second application we investigated is Lucene [6], an open-source text search engine library that is part of the Apache project and is implemented in Java. When Lucene produces the results of a search query, each item is given a relevance score, and the results are sorted accordingly. With a known, finite set of documents to search, it is trivial to check whether the items in the query result really do contain the query keywords (and that documents not in the result do not contain the keywords), but the relevance scores and the sorted ordering cannot be known in advance (if they could, there would be no need to have the tool).

Lucene supports simple queries like “romeo”, “romeo AND juliet”, “romeo OR juliet”, “romeo NOT juliet”, etc. It is also possible to boost the relevance scores associated with a particular term, e.g., “romeo^4 juliet” would boost the score of “romeo” four times as much as it does of “juliet”.

The third application investigated in this study is gaffitter [10], which is an open-source application that uses a genetic algorithm to arrange an input list of files and directories into volumes of a certain size capacity, such as a CD or DVD, in a way that the total unused (wasted) space is minimized. This is an example of optimization software, specifically for solving the “bin-packing problem,” which is known to be NP-hard.[1] Solutions to

[1] http://en.wikipedia.org/wiki/Bin_packing_problem

optimization problems in the domain of NP-hard qualify as "non-testable programs": after all, by definition, verifying whether a solution is correct is just as hard as finding the solution in the first place.

The gaffitter software is implemented in C++ and allows the user to specify the list of files and the target size (i.e., the size of each bin). The user can also specify other options related to the genetic algorithm, such as the number of generations, crossover probability, mutation probability, etc. In the first generation of the genetic algorithm, gaffitter randomly creates a number of candidate solutions (collection of files), and measures their quality by calculating how close the solution comes to the target. In subsequent generations, the best solutions are randomly mutated and/or combined with other candidate solutions, and those with the highest quality are carried over to the next generation, while lower-quality solutions are deselected. Once the specified number of generations is reached, or an ideal candidate solution is found, the program concludes. Although genetic algorithms necessarily rely on randomization, gaffitter allows the user to specify a seed for the random number generator so that the results would be deterministic; we used a seed of 0 in our experiments.

Methodology

We used mutation testing in this experiment, just as in the one previously described.

For JSim, we were only able to generate six mutants that could be used in the experiment. These were all in the Task class, which includes code responsible for tracking the amount of time spent on each step in the simulated process. The two principle reasons why we were only able to create a small number of mutants were that there simply are not many mathematical calculations in JSim, and that many of the mutants we generated led to obvious errors, such as crashing. We investigated the use of a mutant generator tool such as μJava [112], which would also create other types of mutations specific to Java (such as modifying inheritance hierarchies, variable scope, etc.) but the current implementation

of μJava (version 3) as of this writing does not support Java generics[2], which are used throughout the JSim implementation. We also considered simulating defects in JSim through modifications to the configuration files, but such a technique would not be useful in measuring the effectiveness of runtime assertion checking, which aims to find code defects. Other non-systematic approaches such as manually inserting defects could introduce threats to validity if the defects are biased towards (or against) a particular testing approach. Thus, even though six mutants is a small number, we feel that this is the only fair way to compare the testing techniques. Of the six mutants, one was a mathematical operator and five were off-by-one errors.

For Lucene, we only placed mutations in the DefaultSimilarity and TermScorer classes, because they are the ones that do most of the work in the simple searches we tested (note that as Lucene is a large framework for building search applications, many of the classes are not useful for our purposes). Also, none of the mutations of the comparison operators (e.g., changing "less than" to "greater than") were usable because they all caused obvious failures, such as the application crashing or producing no results. Thus, we only had 15 mutants to use in the experiment: four mathematical errors and 11 off-by-one errors.

For gaffitter, we created a total of 66 mutants: 15 related to comparison operators, 19 related to mathematical operators, and 32 off-by-one errors.

Creating Partial Oracles. As in the first experiment, for each application we created simple test cases whose correct output could easily be calculated. To ensure that these simple test cases are not *too* simple, we measured the line coverage of the programs when processing these inputs, and checked that the coverage is approximately the same as when using the arbitrary inputs for metamorphic testing and runtime assertion checking.

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

[2]http://cs.gmu.edu/ offutt/mujava/

For the Lucene partial oracle, we manually created a simple document corpus in which the first file consisted of only the word "one", the second contained "one two", the third contained "one two three", and so on. Thus, we could easily predict not only the results of queries like "one AND three" but also know the expected relevance ranking. Note that we could not predict the scores themselves, only their relative values; the scores do not have any meaning on their own.[3]

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

Metamorphic Testing. Using the guidelines described previously for conducting metamorphic testing, we assessed each of the three applications and enumerated a set of properties that were verified using the gold standard implementation of each program. Then, for each mutant, we determined whether there was a violation of the metamorphic property when running the program on arbitrary test input; if so, the mutant was killed and the defect had been found.

For JSim, we identified two metamorphic properties. First, because the event timings as specified in the configuration have no time units, multiplying each by a constant value should effectively increase the overall time to complete the process by the same factor. For instance, if there are N events in the process, and the time to complete each step is t_0, t_1, t_2... t_{N-1} respectively, and the total time to complete the process is observed to be T, then changing the event timings to $10t_0$, $10t_1$, $10t_2$... $10t_{N-1}$ would be expected to change the overall process time to $10T$.

Second, we considered the effect that changing the event timings would have on the

[3] http://article.gmane.org/gmane.comp.jakarta.lucene.user/12076

utilization rates for the different resources in the simulation. Specifically, if the event timings were increased by a positive constant, then the utilization rate (i.e., the time the resource is used, divided by the overall process time) of the most utilized resource would be expected to decrease, since the total increase in the overall process time would outweigh the small increase in the resource's time spent working. As a simple example, consider a process that has steps that take times a, b, and c to complete, and a resource that is used in the third of these steps. Its utilization would thus be $c/(a + b + c)$. If the time for each step were increased by one, then the utilization rate would change to $(c + 1)/(a + b + c + 3)$. If this resource has the highest utilization, i.e., $c > a > 1$ and $c > b > 1$, then the new utilization rate will be lower, because the change to the numerator is proportionally smaller than the change to the denominator.

In the metamorphic testing of JSim, we used the same data set used in the emergency room simulation work presented by Raunak et al [156].

The metamorphic properties for Lucene are based on those presented by Zhou et al. [203], in which the authors applied metamorphic testing to Internet search engines such as Yahoo!, Google, and Microsoft Live Search. In addition to the properties presented in that work, we also included others based on the specific syntactic features of Lucene, as described in its user manual. The properties are listed in Table 3.13. The corpus used for Lucene was the text of Shakespeare's "Romeo and Juliet".[4]

ID	Property	Description
L1	romeo OR juliet == juliet OR romeo	Commutative property of boolean operator
L2	romeo julietˆ0.25 == romeoˆ4 juliet	Boosting one term by a factor of 4 should be the same as boosting other term by a factor of 0.25
L3	romeo OR foo == romeo	Inserting an OR term that does not exist should not affect the results
L4	romeo NOT foo == romeo	Excluding term that does not exist should not affect the results

Table 3.13: Metamorphic Properties Used for Testing Lucene

[4]http://shakespeare.mit.edu/romeo_juliet/index.html

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

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

Invariant Detection. To create the set of invariants that we could use for runtime assertion checking, we applied Daikon to the "gold standard" (i.e., with no mutations) of the three applications. We executed the applications multiple times, with the different data sets described above. This was done in order to reduce the number of "spurious invariants" that Daikon might produce.

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

The complete set of Daikon-generated invariants is listed in Appendix A of the tech report [128] describing this study.

Pseudo-Oracles. As in the previous experiment, we were unable to obtain sufficient pseudo-oracles that could be used in the study. For JSim, we intended to use the simulation

tool Arena from Rockwell Automation [171], but were told by the developers of JSim that there were known slight differences between the implementations that would affect the results [184]. For Lucene, we could not find any other search engine that used the same scoring algorithm, thus introducing the possibility of differences that were not a result of defects in the code. Last, although there certainly are other implementations of genetic algorithms that attempt to solve the bin-packing problem, none of the others that we investigated produced the exact same results as gaffitter, because of implementation issues related to the use of random numbers and seeds for the generators.

Results

Table 3.14 shows the effectiveness (percentage of distinct defects found in each application) for the partial oracle, metamorphic testing, and assertion checking approaches for the three applications.

For JSim, all six defects were detected by metamorphic testing and runtime assertion checking, compared to only four when using the partial oracle. For Lucene, both the partial oracle and metamorphic testing discovered 11 of the 15 defects; assertion checking only detected nine. Last, for gaffitter, metamorphic testing killed 22 of the 66 mutants, whereas the partial oracle killed only 14 and assertion checking killed 20.

App.	# Mutants	Partial Oracle	Metamorphic Testing	Assertion Checking
JSim	6	4 (66.7%)	6 (100%)	6 (100%)
Lucene	15	11 (73.3%)	11 (73.3%)	9 (60%)
gaffitter	66	14 (21.2%)	22 (33.3%)	20 (30.3%)

Table 3.14: Distinct defects detected in Study #2.

Discussion

Because of the small number of mutations used in the experiment for JSim and Lucene, the results may not be statistically significant, but serve to demonstrate that metamorphic

testing is at least as effective as the other approaches. This may not be an important finding in and of itself, but combined with the results of the first experiment, and considering the results for gaffitter, indicates that metamorphic testing is no less effective than the current state of the art at finding defects in a variety of applications without test oracles.

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

For instance, consider the pseudo-code in Figure 3.7, which summarizes lines 212-226 in GeneticAlgorithm.cc and is typical in most genetic algorithms. This function takes two candidate solutions (i.e., sets of items) called CS1 and CS2, and creates a child candidate solution, which contains some items from CS1 and some from CS2. On line 2, a random number is used to determine whether CS1 and CS2 should cross over at all. If so, on line 3, a crossover point (i.e., an index in the list of items) is randomly chosen. In line 4, the first items (up to the crossover point) of CS1 are merged with the last items (from the crossover point to the end) of CS2.

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

Figure 3.7: Sample code from genetic algorithm

Considering the mutations such as those we inserted into gaffitter, it is clear that many

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

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

This leads to an interesting question regarding whether the mutations we put into the code are actually "defects" in all cases. The mutated versions we used in this experiment are ones for which the program output differed from that of the gold standard, i.e., the list of selected files was not the same. However, in some cases the output in the mutated version had the same quality as that of the gold standard. Does this mean that the output is "incorrect"? One could argue that there is no practical difference in that there can be multiple solutions, all of which are equally good. For the sake of this experiment, though, we will consider these differences to be defects, using the definition from Section 1.1, since the output deviates from what is expected according to the gold standard, which we assume to be error-free. The justification is that these defects may not have affected the quality of

the output for these particular cases, but might for others.

Note that we were only able to identify two metamorphic properties for this application. This is partly because for many of the inputs to a genetic algorithm, the effect of changing them cannot be known in advance. For instance, one cannot predict the changes to the output based on changing inputs like crossover rate, mutation rate, etc. After all, if it could be known that a particular value would have a positive effect on the overall result quality, then there would be no need to vary that input. The only inputs for which we could predict changes to the outputs were the list of files, the target size, and the number of generations; these were all used in the metamorphic properties we considered. If we were able to identify more, it follows that more mutants may have been killed by further testing. As in the case of PAYL in the study described in the previous section, if we were able to conduct the metamorphic tests *inside* the application at the function level, we may be able to identify more properties and run more tests; such an approach is described in Chapter 4.

Analysis of Metamorphic Testing Results

Here we analyze the effectiveness of the different metamorphic properties used for the testing of the three applications.

JSim. Table 3.15 shows the effectiveness of the two metamorphic properties used in the testing of JSim. The first (labeled "Multiplication") involves multiplying all event timings by a constant factor; the total overall time should also increase by that factor. The second (labeled "Addition") involves adding a constant to all event timings; the utilization rate of the most utilized resource should decrease.

Mutation	**# Mutants**	**Multiplication**	**Addition**	**Total**
Math operators	1	1	1	1 (100%)
Off-by-one	5	5	2	5 (100%)
Total	**6**	**6**	**3**	**6 (100%)**

Table 3.15: Results of mutation testing for JSim using metamorphic testing

The most interesting result here is that, unlike in the previous experiment with the machine learning applications, the metamorphic property based on multiplication was very effective at killing the off-by-one mutants. Upon further investigation, we see that this is due to the nature of the particular way in which the off-by-one mutant is manifested.

Consider a simple function $f(a, b) = a + b$, and a mutated version with an off-by-one error $f'(a, b) = a + b - 1$. We would expect f to exhibit the metamorphic property that $f(10a, 10b) = 10f(a, b)$; obviously, f' does not exhibit this property, since $f'(10a, 10b) = 10a + 10b - 1 = f'(a, b) - 1$. In this case, the property based on multiplication reveals the defect. The code we mutated in JSim included such simple functions, thus the property was very effective.

On the other hand, in the machine learning applications tested in the first experiment, the off-by-one mutants were more likely to appear in loop control structures, array indexing, etc. In Figure 3.6 above, in which there is an off-by-one error in the implementation of bubble sort, the result is that the ordering of the values is incorrect. However, the values *themselves* are not affected by the defect, so a metamorphic property based on multiplying them by a constant will not be violated: the ordering of the values will still be as expected, even though it is incorrect.

Lucene. Table 3.16 shows the effectiveness of each of the four metamorphic properties (listed in Table 3.13) for Lucene.

Mutation	**# Mutants**	**L1**	**L2**	**L3**	**L4**	**Total**
Math operators	4	0	0	2	0	2 (50%)
Off-by-one	11	0	1	6	2	9 (81.8%)
Total	**15**	**0**	**1**	**8**	**2**	**11 (73.3%)**

Table 3.16: Results of mutation testing for Lucene using metamorphic testing

Property L3 is clearly the most effective. This property declares that the results from the query “romeo OR foo” should be the same as those from the query “romeo”, since the word “foo” does not appear in any of the text, and should not affect the results. The reason

why this property is effective is quite clear: in determining the quality of each search result, the term "foo" is being given a non-zero score because of the mathematical and off-by-one errors, thus the total score changes, even though it should not.

On the other hand, property L1 is not effective because, even in the presence of the defects, the queries "romeo OR juliet" and "juliet OR romeo" will return the same scores, even if they are incorrect.

gaffitter. Table 3.17 shows the effectiveness of metamorphic testing for detecting each of the types of mutants used in the experiment, broken down by the different metamorphic properties, where "G1" refers to increasing the number of generations, and "G2" refers to multiplying the file sizes and target size by a constant.

Mutation	**# Mutants**	**G1**	**G2**	**Total**
Comparison operators	19	7	3	7 (36.8%)
Math operators	15	8	2	8 (53.3%)
Off-by-one	32	7	2	7 (21.9%)
Total	**66**	**22**	**7**	**22 (33.3%)**

Table 3.17: Results of mutation testing for gaffitter using metamorphic testing

Analysis of the mutants that were killed reveals that most of them were in the code that selects the top N candidate solutions (where N is configurable) to be used in the next generation. If this code has defects, then some candidate solutions that are of lesser quality may survive whereas ones with higher quality may be removed; thus, the quality of future generations may decline, in violation of the metamorphic property.

On the other hand, for defects in other parts of the code, this particular metamorphic property was not effective at killing the mutants. For instance, if the calculation of the quality of a candidate solution were incorrect, the property would not be violated as long as the best candidate solutions survived into future generations, regardless of the correctness of the quality calculation. Similarly, as described previously, mutants in the code that performs

crossovers between candidate solutions are not likely to have much impact on the overall quality; thus, the quality will likely continue to improve over subsequent generations, even if the implementation of crossing over contains an error.

Clearly these shortcomings could be addressed with the identification of more metamorphic properties. In Chapter 4, we will see that identifying properties at the function-level, as opposed to the system-level, may be more useful in detecting errors related to these calculations.

3.4.4 Study #3: Non-Deterministic Applications

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

Test Subjects

In our first two experiments described above, all the applications were deterministic, and thus none of the defects found were a result of non-determinism. In our next experiment, we apply the testing approaches to non-deterministic applications to measure the effectiveness of each technique.

JSim. The JSim [193] discrete event simulator (described in the previous experiment)

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

MartiRank. The MartiRank application described above also can be non-deterministic. In its second ("ranking") phase, MartiRank performs sorting on the elements in the testing data to achieve the final ranking according to the given model. If the data set contains missing values (which is very likely to happen when using real-world data [127]), MartiRank will randomly place the missing elements throughout the sorted list. Thus, subsequent executions may yield different results, making the output non-deterministic. Note that in our first study, the MartiRank data sets had no missing values, thus the output was deterministic.

Methodology

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

For MartiRank, we were able to use mutation testing as described in the first two experiments and inserted a total of 59 defects (18 mathematical operator mutants and 41 off-by-one mutants). Note that these are not the same set of mutants used in Study #1, since in that study we considered the training phase of MartiRank (as it constructs the model) and here we consider its ranking phase (when it applies the model to testing data). There is some overlap in the code, but other parts are completely different.

Note also that we did not include pseudo-oracles for this study, as the one for MartiRank

was found to be unsuitable, as described in Study #1.

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

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

Metamorphic Testing. In this experiment, we used statistical metamorphic testing (SMT), which has been proposed by Guderlei and Mayer as a technique for testing non-deterministic applications that do not have test oracles [71] and is summarized briefly in Section 3.3. SMT can be applied to programs for which the output is numeric, such as the overall event timing in JSim, and is based on the statistical properties of multiple invocations of the program. That is, rather than consider the output from a single execution, the program is run numerous times so that the statistical mean and variance of the values can be computed. Then, a metamorphic property is applied to the input, the program is again run numerous times, and the new statistical mean and variance are again calculated. If they are not as expected, then a defect has been revealed.

For JSim, we specified non-deterministic event timing over a range $[\alpha, \beta]$ and ran the simulation used in the previous experiment 100 times to find the statistical mean μ and variance σ of the overall event timing in the process (i.e., the time to complete all the steps). We then configured the simulator to use a range $[10\alpha, 10\beta]$, ran 100 more simulations, and expected that the mean would be 10μ and the variance would be 10σ. Of course, they results would not *exactly* meet those expectations, so we used a Student T-test to see if any

difference was statistically significant; if so, then the defect was revealed. We validated this approach with the gold standard implementation (i.e., in which we had not inserted any defects), and found that the resulting distributions were not significantly different, with $p < 0.05$.

Heuristic Metamorphic Testing (HMT), a new technique introduced above in Section 3.3, is a similar approach that is based on SMT but can be used for non-deterministic applications or functions that produce non-numerical output, such as the ranking of testing data examples in MartiRank. For such applications, certain metamorphic properties can be applied so that the new output is expected to be "similar" to the original, where the expected similarity is determined by observing multiple runs of the program.

In this experiment, we ran 100 executions of MartiRank with a given input, which produced a ranked list of elements. We could then compare the similarly of those lists of elements using the Spearman Footrule Distance [173], which yields a normalized equivalence rating of how similar the lists are, given that they are always expected to contain the same elements. We then permuted the input (by changing the ordering of the examples in the data set for MartiRank), ran 100 executions with the new input, and calculated the normalized equivalence of the new outputs. As with SMT, we again used a Student T-test to compare the results, and we validated this approach using the gold standard implementation.

Note that just as metamorphic testing is essentially a pseudo-oracle approach, in that "expected results" are not necessarily "correct results", SMT and HMT can only indicate defects, not correctness: if the statistical mean and variance are not as expected (in SMT) or if the normalized equivalence is not as expected (as in HMT), that indicates that something is amiss; but if the results *are* as expected, a defect may still exist but not be revealed.

Invariant Detection. As in the previous study, we applied Daikon to the applications and had it observe multiple program executions of the "gold standard" implementation so that it could create a list of invariants. The mutated versions of the software were then

executed with the same data sets to see whether any invariants were violated. Because of the non-deterministic nature of the applications, each was run 100 times, since some invariants might only be violated occasionally.

The complete set of Daikon-generated invariants is listed in Appendix B of the tech report [128] describing this study.

Results

To determine the effectiveness of the three techniques, we applied each to all of the mutated variants of JSim and MartiRank. Table 3.18 shows the number of defects found by the three techniques in this study. The metamorphic testing approaches were most effective in both cases.

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

Table 3.18: Distinct Defects Found in Study #3.

Discussion and Analysis

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

elements are out of place and the normalized equivalence will be lower), and the defect will be revealed.

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

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

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

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

independent executions.

3.4.5 Summary

We now revisit the research questions that this experiment was designed to answer.

1. Is system-level metamorphic testing more effective than other techniques for detecting defects in applications without test oracles, particularly in the domains of interest?

2. Is Heuristic Metamorphic Testing more effective than other techniques for detecting defects in non-deterministic applications without test oracles?

3. Which metamorphic properties are most effective for revealing defects in the applications of interest?

For questions #1 and 2, the answer is clearly "yes". In the first two studies, metamorphic testing killed 217 of the 269 total mutants (80.6%), whereas assertion checking killed 188 (69.8%) and the partial oracle killed 176 (65.4%). In the third study, the metamorphic testing approaches killed 59 of the 78 total mutants (75.6%) in the non-deterministic programs, whereas assertion checking killed 27 (34.6%) and the partial oracle killed 15 (19.2%).

For question #3, properties based on permuting or negating the input were most effective for detecting the types of defects we inserted into the machine learning applications evaluated in the first study. Properties based on multiplication and addition were considerably less effective.

3.4.6 Threats to Validity

Some of the details of the three studies may cause the reader to wonder how generalizable the results are.

The programs that were investigated may not be representative of all programs that do not have a test oracle. However, these were selected so as to demonstrate that metamorphic testing is applicable to a range of application domains that do not have test oracles: specifically, machine learning (C4.5, MartiRank, SVM, PAYL), discrete event simulation (JSim), information retrieval (Lucene), and optimization (gaffitter). Moreover, within the domain of machine learning, we chose applications from different subdomains, specifically decision tree classifiers (C4.5), linear classifiers (SVM), ranking (MartiRank), and unsupervised machine learning (PAYL). Given the number of test subjects and the breadth of application domains, plus the fact that many of these applications are (or are part of) industrial systems, we feel that the results can safely be generalized. However, although we investigated multiple supervised machine learning applications, future work could investigate the effectiveness of detecting defects in other unsupervised applications beyond PAYL.

Another issue is related to the types of defects that were planted in the applications via mutation testing, and whether or not the ability to detect mutants is a fair metric for comparison of testing approaches. As indicated above, though, mutation testing has been shown to be an accurate approximation of real-world defects [3], and is generally accepted as the most objective mechanism for comparing the effectiveness of different testing techniques [55, 167]. Additionally, we assumed in the mutation testing experiment that the version we used as the gold standard was free of defects. If this were not true, then the invariants created by Daikon may not all have been sound, and some sound invariants may have been omitted; additionally, defects supposedly revealed by any of the testing approaches may have been the result of detecting other defects, not the ones inserted via mutation. However, we felt that the likelihood of defects (or, at least, defects that would influence the results) existing in the gold standard was very low, given that: we were able to verify that the expected metamorphic properties held in the gold standard implementation for all applications and all test inputs; and, the partial oracle for the gold standard produced the expected result for all applications and all test inputs.

A third potential threat to validity involves the data sets that were used in the experiments. For the partial oracles, the data sets were typically smaller than the data sets used for metamorphic testing and runtime assertion checking. Of course, the data sets were smaller out of necessity because, for larger data sets, the correct output could not easily be predicted, thus they could not be used as partial oracles. Also, as noted, the line coverage of the data sets was approximately the same for all the testing approaches, and variations of the original partial oracle data sets were generated to attain a better mix of values. Last, for all experiments but particularly Study #1, the data sets were selected because they were readily available to us; it is possible that the results may have been different had different data sets been chosen, but with very few exceptions, we did not see any single data set being particularly good (or particularly bad) at revealing defects, and the results would not have changed significantly had any data set been omitted.

In these studies, we used Daikon to create the program invariants that would be used in runtime assertion checking. Although some researchers have questioned the usefulness of Daikon-generated invariants compared to those generated by humans [153], Daikon is generally accepted as the state-of-the-art in automatic invariant detection [141]. We chose to use the tool so that we could eliminate any human bias or human error in creating the invariants, which would require a great deal of knowledge of the source code, considering the complexity of the applications we evaluated. Hu et al. [85] performed a study in which invariants were hand-generated, but the programs they investigated were considerably smaller than the ones we used in our experiments, and we did not feel that such an approach would scale to larger, more complex systems.

As noted above, Daikon can generate spurious invariants if there are not enough program executions, but we mitigated this by running the programs multiple times with different inputs, and by ignoring any invariants that were obviously a result of the data sets that were being used. Note that even if some of the invariants that detected defects in the study actually were spurious, this would only serve to increase the apparent effectiveness of

runtime assertion checking. Given that our intent was to show that metamorphic testing is more effective, any experimental error related to spurious Daikon invariants would, in fact, only strengthen that claim.

Clearly the ability of metamorphic testing to reveal defects is dependent on the selection of metamorphic properties, and the results may have varied had we selected different ones instead. However, it was our intention to show that even the basic metamorphic properties described in Section 2.4 can be used even without a particularly strong understanding of the implementation, and we have shown that these are shared by many applications in the domains of interest (Section 2.5). Using this approach, therefore, we are demonstrating the *minimum* effectiveness of metamorphic testing; the use of application-specific properties may actually reveal even more defects.

Last, as acknowledged previously in introducing this set of experiments, we did not compare our results to those of techniques such as formal specification languages or trace and log file analysis. A qualitative discussion of how metamorphic testing techniques compare to the approaches surveyed by Baresi and Young [15] can be found in the Related Work section (Section 6.1).

3.5 Summary

In this chapter, we have presented a testing framework called *Amsterdam*, which addresses some of the practical limitations of metamorphic testing so that it can be an efficient technique for testing applications that deal with large, complex data sets.

Additionally, we have introduced a new technique called *Heuristic Metamorphic Testing*, which can be used to test non-deterministic applications. The application is run multiple times to build a profile of the outputs, using a domain-specific heuristic, and then the metamorphic transformation is applied: the application is then run multiple times again, and a measurement is taken to see if the profile of the new outputs is statistically similar to

what is expected. If not, then a defect has been detected.

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

By showing metamorphic testing to be more effective than other approaches for testing programs that have no oracle, we have proven the hypothesis (stated in Section 1.6) that "for programs that do not have a test oracle, automating the process of metamorphic testing advances the state of the art in detecting defects." In each of the three studies, metamorphic revealed more defects on average than using partial oracles or runtime assertion checking using Daikon-detected program invariants, and was particularly effective for finding defects in non-deterministic applications.

In the next chapter, we seek to improve the metamorphic testing technique by checking the metamorphic properties of *individual functions*, as opposed to just those of the entire system, and to determine whether this approach will be more effective at detecting defects.

Chapter 4

Metamorphic Runtime Checking

Although the empirical studies in the previous chapter demonstrate that metamorphic testing is very effective at detecting defects in applications without test oracles, we noticed two important inherent limitations. One is that the application itself may not have a sufficient number of metamorphic properties, possibly because of rigorous restrictions placed on the allowable input space (this was an issue with the intrusion detection system PAYL) or the small number of system inputs (as in the genetic algorithm gaffitter). Another limitation is that metamorphic testing at the system level may not detect defects in individual functions that do not have much effect on the overall output in a way that violates the property, as was the case for gaffitter.

To address these limitations, and to improve the effectiveness of metamorphic testing, we suggest that the checking of metamorphic properties could also occur *inside* the application, at the function level. That is, in addition to checking the system-level metamorphic properties of the entire application as a whole, we also intend to check the metamorphic properties of *individual functions* as the software is running. As opposed to performing system testing based on properties of the entire application, or by conducting unit testing of isolated pieces of code, in this chapter we present a technique for testing applications that do not have test oracles by checking the metamorphic properties of its individual functions

as the full application runs. This will allow for more fine-grained testing that is not as restricted by limitations on the input space, and also makes it possible to check for subtle defects inside the code that do not have a large impact on the overall output.

In this chapter, we introduce a new type of testing called *Metamorphic Runtime Checking*. This is a system testing technique in which, rather than specifying the metamorphic properties of the application as a whole, we do so for individual functions. While the program is running, we apply functions' metamorphic properties to derive new test input for those functions, so that we should be able to predict the corresponding test output; if it is not as predicted, then there is a defect in the implementation. We also present an implementation framework called *Columbus* that supports the execution of Metamorphic Runtime Checking from within the context of an application as it runs, so that program inputs can be used to drive the arguments used in metamorphic testing of the individual functions.

We then describe the results of new empirical studies of real-world programs without test oracles to demonstrate the effectiveness of Metamorphic Runtime Checking, and show that conducting metamorphic testing based on the properties of individual functions is able to reveal defects not found by metamorphic testing based on system-level properties alone.

The rest of this chapter is organized as follows: in Section 4.1, we introduce the Metamorphic Runtime Checking approach, including the necessary steps to be performed. In Sections 4.2 and 4.3, we describe the model and architecture of the Columbus implementation framework, respectively. We present the results of feasibility studies in Section 4.4, and in Section 4.5 we discuss empirical studies that measure the effectiveness of Metamorphic Runtime Checking, and compare the results to metamorphic testing at the system level. In Section 4.6 we consider the performance cost of testing software using Metamorphic Runtime Checking.

4.1 Approach

Metamorphic Runtime Checking is a system testing technique based on checking the metamorphic properties of individual functions, rather than just those of the entire system, as the application is running. Metamorphic Runtime Checking can be considered a cross between metamorphic testing and runtime assertion checking [43]. The metamorphic properties that are checked are analogous to program invariants: at the point in the program execution in which the function is called, its metamorphic properties are expected to hold. If the metamorphic property is violated, then a defect has been revealed.

This technique is orthogonal to the techniques described in Chapter 3, in that the various techniques can be combined to ensure that the properties of *both* the entire system *and* of individual functions hold as the program executes. That is, Metamorphic Runtime Checking does not preclude the checking of metamorphic properties at the system level. However, in the rest of this chapter, the two techniques will be discussed and compared separately.

4.1.1 Overview

The Metamorphic Runtime Checking approach entails four steps. Note that by "tester" we mean "the person who is testing the code"; this may include the original author (developer) of the code, or a separate third-party test engineer.

1. **Identify metamorphic properties.** The tester must first identify the metamorphic properties of the functions that will be used in the testing. This step is further described in Section 4.1.2.

2. **Specify metamorphic properties.** For each function to be tested, the tester specifies its metamorphic properties using a notation based on the Java Modeling Language (JML) specification language [98], further described below in Section 4.3.1.

3. **Convert the specifications into tests.** In this step, described in Section 4.3.2, the

tester uses a preprocessor to convert the specifications into test functions. These tests will automatically be added to the original source code.

4. **Conduct system testing.** Once the metamorphic properties are specified and the code is instrumented with the tests, system testing can commence. The properties are checked as the individual functions execute, and any test function outputs that deviate from what is expected are indicative of defects in the code. Unlike in conventional unit testing, the tester need not construct any specific test harness; rather, the functions are effectively tested by simply executing the entire program.

The approach is not limited only to "pure" functions that do not have side effects, nor to metamorphic properties that depend only on a function's formal parameters as input and its return value as output. In fact, functions need not have any input parameters or any return values to have metamorphic properties: the properties can just specify the expected relationship between the system state before the function call and the state after the call. Consider a function to calculate standard deviation function, which operates on an array that is part of the system state, and has no return value, but rather updates a global variable as a side effect. The function's metamorphic property can still be checked by considering the array as the "input" and the value of the global variable as the function's "output".

Because Metamorphic Runtime Checking must allow the functions under test to be run multiple times, it must therefore permit these functions to have side effects, but ensure that the side effects of the additional invocations do not affect the process' system state after the test is completed. Clearly, changes to the state caused by calling the function again with modified inputs would be undesirable, and may lead to unexpected system behavior later on. Thus, the metamorphic tests are conducted in a "sandbox", as described below in Section 4.2.

4.1.2 Devising Metamorphic Properties

As discussed in Section 2.4, an open issue in the research on metamorphic testing is, "how does one know the metamorphic properties of the functions?" Typically it is assumed that the tester will be familiar enough with the algorithm being implemented so that he or she can identify these properties. Although this is generally a reasonable assumption for system-level metamorphic properties, the conventional wisdom is that determining the metamorphic properties of individual functions requires much more detailed knowledge of the code.

In our experience with Metamorphic Runtime Checking, we noted anecdotally that the metamorphic properties of the entire application were often also reflected in one or more individual functions within the code. That is, if one can identify a property of the application, then it is often the case that there will be a function (or perhaps even more than one) that exhibits the same property.

Investigation of this phenomenon is outside the scope of this particular work, but during our experiments we observed that often there would be data structures that represented the program input data (either all of it, or a significant part of it); any function that took such a data structure as a parameter was likely to exhibit the same metamorphic property as the entire application since, essentially, the input to the function was the same as the input to the program. For instance, the SVM implementation in Weka (described in Section 2.5.3) contains a method "buildClassifier" which takes as input the set of examples from the training data and constructs the set of support vectors. And C4.5 (Section 2.5.4) has a function called "FormTree" that considers a group of examples and builds the decision tree. Both of these essentially wrap all of the functionality of the program as a whole, and thus have the same metamorphic properties.

Of course, the Metamorphic Runtime Checking of such properties may not be particularly useful, since the properties can simply be checked with system-level metamorphic testing. In terms of looking for other metamorphic properties (not necessarily those of the

entire application), we also observed that many of the applications in the domain of interest (specifically, machine learning) include "pure" functions, i.e., those that do not depend on, nor alter, the system state. These functions often take numerical or well-structured data as input and produce similar output; functions like sorting or performing a calculation fall into this category. Such functions are good candidates for exhibiting the metamorphic properties classified in Section 2.4.1 because they are essentially mathematical, and exhibit well-known algebraic properties such as distributivity and transitivity [133].

It is our assumption that, in practice, the software architect or developer would be able to more easily identify the functions' metamorphic properties, since they will have more familiarity with the code than a third-party tester does. The same guidelines prescribed in Section 2.4 could be applied here, as well.

Last, we note that test case selection (i.e., choosing the test cases most likely to reveal defects) in metamorphic testing is detailed elsewhere [34] but is not further described here. In Metamorphic Runtime Checking, all specified metamorphic properties are tested whenever the corresponding function is called, in whatever states and for whatever inputs that are encountered during execution.

4.2 Model

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

For instance, for a function to calculate the standard deviation of an array of numbers, whenever the function is called, its argument can be passed along to a test method, which

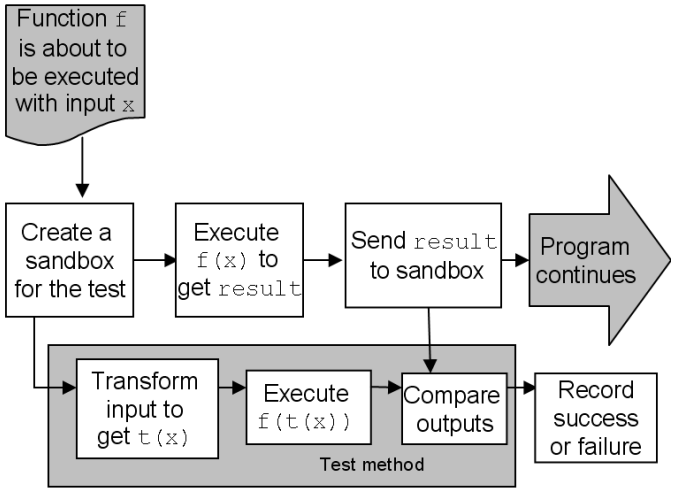

Figure 4.1: Model of Metamorphic Runtime Checking

will multiply each element in the array by -1 and check that the two calculated output values are equal; at the same time, another test method can multiply each element by 2 and check that the new output is twice as much as the original. It is true that if the new output is as expected, the results are not necessarily correct, but if the result is *not* as expected, then a defect must exist. This model will allow us to execute tests within the context of the running application, in applications without a test oracle, by using the metamorphic tests themselves as built-in pseudo-oracles.

In our model of the testing framework, metamorphic tests are logically attached to the functions that they are designed to test. Upon a function's execution, the framework invokes its corresponding test(s). 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 4.1 demonstrates the model we will use for conducting these tests.

Metamorphic Runtime Checking must be done in such a manner that any changes to the state of the process are the result of only the main (original) function execution, and not from any function calls that are only for testing purposes. In other words, there must not be any observable modification of the application state; however, the tests themselves *do* need to be able to modify the state because the functions are necessarily being called multiple

times, which could have side effects. Thus, the modifications to the state that are caused by the tests must not affect the application, so that the application can keep executing and testing can continue.

One solution is to run the tests in the same process as the user state and then transactionally roll them back (an idea explored by Locasto et al. [107]). Another approach is to create a "sandbox" so that the test function runs in a separate cloned process that does not affect the original; the sandbox must also make sure that the test function does not affect external entities such as the file system. The current implementation of the Metamorphic Runtime Checking framework uses the sandbox approach, as further described below in Section 4.3.2.

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

4.3 Architecture

In order to facilitate the execution of Metamorphic Runtime Checking, we require a framework that conducts the function-level tests during actual runs of the application, using the same internal state as that of the original function. A system like Skoll [121] 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 by Clarke and Rosenblum [43]) or monitoring tools (such as Gamma [144]) could be used, but they generally do not allow for calling a function again with different arguments (which we require), and do not safeguard against visible side effects. Thus, a new solution is required.

In this section we introduce the *Columbus* testing framework, implemented for appli-

cations in both C and Java. The Columbus framework allows software testers to specify metamorphic properties, instrument the application code, and conduct metamorphic tests at the function level as the program executes.

4.3.1 Creating Tests

For the functions that are to be tested, the Columbus framework must be provided with executable test code that specifies the metamorphic properties to be checked within the running program. Because the instrumentation of the functions is done at compile-time, we currently assume access to the source code. Given that it is the software developers and testers who will write the tests and instrument the code, we feel that this assumption is reasonable. However, as it may not always be possible or desirable to recompile the code, an approach to dynamically instrumenting the compiled code, such as in Kheiron [69], could be used instead.

To aid in the generation of these tests, we have created a pre-processor to allow testers to specify metamorphic properties of a function using a special notation in the comments [133]. As in JML and many other specification languages, the properties are specified in annotations in the comments preceding the function with which they are associated, or in a separate file. In our notation, the metamorphic properties are specified in a line starting with the tag "@meta" and then are followed by a boolean expression that states the property.

Figure 4.2 shows such properties for an implementation of the sine function in C, which exhibits two metamorphic properties: $\sin(\alpha + 2\pi) = \sin(\alpha)$ and $\sin(-\alpha) = -\sin(\alpha)$. The parameter "\result" represents the return value of the original function call, so that outputs can be compared; this notation is typical in specification languages such as JML. These properties can then be used by the pre-processor in the testing framework to generate the test code shown in Figure 4.3.

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

Figure 4.2: Specifying metamorphic properties

```
int __MRCtest_sine(double angle, double result) {
  if (( sine(angle + 2 * M_PI) == \result) == 0) return 0;
  if (( sine(-1 * angle) == -1 * \result) == 0) return 0;
  return 1;
}
```

Figure 4.3: Example of a Metamorphic Runtime Checking test generated by the pre-processor

Comparing floating point values

As it is written, the above example in Figure 4.3 may fail even if the function is working correctly, due to imprecision in floating point calculations. In order to address this, the notation in the Columbus framework allows floating point values to be compared using a built-in tolerance level (set by default to 10^{-6}), 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 compare it to a tolerance, as is customary in most specification languages.

Array functions

To simplify the specification of some of the types of metamorphic properties that we feel would be typical, based on our evaluation in Section 2.4, we have also added special keywords to the notation, using the JML style of starting keywords and operators with a backslash. These allow for the execution of operations on arrays, or for Java applications, on classes that implement the Collection interface that would be used during the test; Table 4.1 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`
`\permute(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 `A`
`\exclude(A , x)`	Removes an element `x` from `A`
`\concatenate(A , B)`	Combines the elements of `A` and `B` into a new array or Collection

Table 4.1: Additional keywords for manipulating arrays

An example of the use of these keywords appears in Figure 4.4. When calculating the standard deviation for an array of integers, permuting 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 standardDev(\permute( A )) == \result;
 @meta standardDev(\multiply( A , 2)) == \result * 2;
 */
 double standardDev (int[] A) { ...  }
```

Figure 4.4: Example of using built-in array functions for specifying metamorphic properties

Conditionals

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 specifying metamorphic properties, using if/else notation. Figure 4.5 shows an example.

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

Figure 4.5: Conditional metamorphic property

Checking within a range

The metamorphic properties of some functions may entail 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, the notation includes two additional boolean functions, as described in Table 4.2; note that either function can used to check that a value is *not* in the range by using the boolean negation operator.

`\in { x ; S }`	Returns true if the value `x` is equal to a member of `S`
`\inrange { x ;` x_1 `;` x_2 `}`	Returns true if $x \geq x_1$ and $x \leq x_2$

Table 4.2: Additional keywords for handling ranges of values in specifications

As an example, consider a function in a personal finance application that calculates the amount of income tax a user must pay, according to a number of input criteria such as annual wage income, stock holdings, tax-free investments, etc. Determining a specific metamorphic property based on modifying any of these inputs may be quite complicated: it is probably not always the case that simply doubling the annual wage income will double the amount of income tax owed. However, it might be fair to say (given the particular implementation), that even though the predicted new output will not exactly be doubled, we may be able to specify that the value should not be *less* than the original output, and perhaps

should not be more than 2.5 times that value. Thus, we can specify this metamorphic property as demonstrated in Figure 4.6.

```
/*@
 @meta \inrange { calculate_tax(\multiply(income, 2) ;
             \result ; \result * 2.5 };
 */
 double calculate_tax (double income, double stock, double ...)
 { ...  }
```

Figure 4.6: Example of metamorphic properties specifying a range of values

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

Non-deterministic functions

Functions that rely on randomness present a challenge for Metamorphic Runtime Checking, since the specific output may vary across function executions, and thus it is difficult to create a metamorphic property that will always hold. For instance, consider a function *rand*(A, B) that is meant to return a random number in the range [A, B]. One would expect that *rand*(A + 10, B + 10) would return a number in the range [A + 10, B + 10], of course, because the range would simply be shifted. However, we cannot say that *rand*(A, B) + 10 == *rand*(A + 10, B + 10), because the second invocation of the function will result in a second call to the random number generator, and the output will change.

In Section 3.3, we discussed metamorphic testing approaches based on statistical properties of a non-deterministic application, or using domain-specific heuristics. However, both of these approaches rely on observing the results of many executions, which is not feasible in Metamorphic Runtime Checking. For a given program execution, we cannot ensure that the non-deterministic function will be run a sufficient number of times with the same arguments so that we can get a statistically meaningful result (it is generally

accepted that at least 30 data points are required to achieve statistical significance [102]). An alternative would be to force multiple executions of the function, with its original inputs and with its transformed inputs, but the overhead of doing so may be prohibitive because of the cost of repeatedly creating the sandbox in which to run the test.

The solution implemented in Columbus is to force the function to be deterministic across the two function calls by modifying the random number generator. If it were guaranteed that the same random number (or sequence of random numbers) would be returned in the second function call, then a metamorphic test could check the results by using a deterministic metamorphic property.

For instance, the code in Figure 4.7 shows an implementation of the function *rand* described above. In this example, the metamorphic property $rand(A, B) + 10 == rand(A + 10, B + 10)$ would be expected to hold if the random number generated by Math.random() were the same in each function invocation.

```
int rand(int A, int B) {
  return (int)(Math.random() * (A - B + 1)) + A;
}
```

Figure 4.7: Code to generate a random number in the range $[A, B]$

Rather than modifying the underlying Java or C libraries to support determinism in the random number generator, Columbus provides a library call that can be used instead. Although this involves modifying the source code, we consider this to be less intrusive than modifying the Java virtual machine or operating system kernel. The Columbus pre-processor can be used to change calls to standard random number libraries to use our special library call, which keeps track of return values so that they can be replayed in the test processes.

We point out here that some sources of non-determinism in applications come not from random number generation, but rather from the underlying operating system or virtual machine on which the application relies. For instance, in the JSim simulator described in

the previous chapter, the output trace of events can be non-deterministic across invocations if some events occur in parallel, since each event is executed in its own Java thread, the scheduling of which is non-deterministic. In these cases, since there is no randomization at the function level, "normal" Metamorphic Runtime Checking can be used at the function level (since each function is deterministic), and metamorphic testing based on statistics or heuristics can be used at the system level.

Specifying more complex properties

Metamorphic Runtime Checking is not limited only to those functions that take input values and return an output, nor is it limited to simple metamorphic properties that can easily be expressed or specified using annotations in the comments. Consider a function `calculate_sum` that determines the sum of the elements in an array referred to by a pointer `p`, and stores that value in a variable `sum`. The tester can then write a test function that permutes the elements in `p`, multiplies them by a random number, calls `calculate_sum`, and checks that the value of `sum` is as expected. Figure 4.8 shows how the tester could then specify that the metamorphic property of `calculate_sum` is described in the function `__test_calculate_sum`; the property is simply that `__test_calculate_sum` should return true.

Note that because `__test_calculate_sum` is called after `calculate_sum`, the framework ensures that the variable `sum` will already have been set by the original function call and will have the appropriate result by the time it is accessed in the first line of the test function. Additionally, the test is executed in a sandboxed process, so the tester does not have to worry about the fact that `sum` will be overwritten by the additional invocation of `calculate_sum`.

```
int* p;
int sum;

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

int __test_calculate_sum() {
  int temp = sum; // remember the old value

  // ...  randomly permute elements in p...

  int r = rand();

  // ...  multiply values in p by r...

  calculate_sum(); // call the function again
  return temp == sum * r; // check the metamorphic property

}
```

Figure 4.8: Example of a manually created Metamorphic Runtime Checking test

4.3.2 Instrumentation and Test Execution

Before compiling the source code, the tester uses the Columbus pre-processor to first generate test code from the specifications, and then to instrument each annotated function with its corresponding test.

During instrumentation, functions to be tested are renamed and wrapped by another function, as shown in Figure 4.9, which shows pseudocode for the wrapper of a Java function `f`. When an instrumented function is to be executed, the function is first called with its input arguments (line 8). Then the "wrapped" original function is called, and any return value is stored in a variable called `result` (line 9). The framework then generates a new process as a copy of the original to create a sandbox in which to run the test code (line 10), ensuring that any modification to the local process state caused by the test will not affect execution of the "real" application, since the test is being executed in a separate

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

Figure 4.9: Wrapper of instrumented function

process with separate memory. At this point, because it is not the test process (line 11), the original process continues by returning the result and carrying on as normal (line 17); meanwhile, in the test process, the original input and the result of the original function call are passed as arguments to the test function (line 12). Within that function, the input can be modified and the outputs can be compared according to the metamorphic properties, without having to worry about changes to the application state. Note that the application and the test run in parallel in two processes: the test does not block normal operation of the application after the sandbox is created. Depending on the configuration and the hardware, the test process may be assigned to a separate CPU or core, so as not to further preempt the original process.

In our current implementation of the Columbus framework, we use a process "fork" to create the sandbox, which gives each test process its own memory space to work in, so that it does not alter that of the original process. In our investigations so far, this has been sufficient for our testing purposes. However, to ensure that the metamorphic test does not

make any changes to the file system, we have also integrated Columbus with a thin OS virtualization layer that supports a "pod" (PrOcess Domain) [147] abstraction for creating a virtual execution environment that isolates the process running the test and gives it its own view of the process ID space and a copy-on-write view of the file system. However, whereas the overhead of using a "fork" can be as little as a few milliseconds (see Section 4.6), the overhead of creating new "pods" can be on the order of a few seconds, so they should only be used for tests that actually affect the file system. Testers can indicate that a "pod" is needed for a test via an annotation in the specification of the metamorphic property; future work could consider automatic detection of which tests need to be run in "pods".

When the test is completed, the framework logs whether or not it passed (Figure 4.9 lines 12-13), the process in which the test was run notifies the framework indicating that it is complete so that the framework can perform any necessary cleanup (line 14), and finally the test process exits (line 15).

Note that Metamorphic Runtime Checking does not preclude "traditional" metamorphic testing using system level properties (as described in Chapter 3) in which the entire application itself is also run a second time with transformed inputs, so that system-level metamorphic properties can also be checked once the process has run to completion. This means that both system-level and function-level properties can be checked during execution, increasing the likelihood of detecting defects.

4.3.3 Limitations

The current implementation of the Columbus framework does have some limitations. As pointed out previously, the test functions are called *after* the function to be tested, rather than at the same point in the program execution. This limitation grew out of the necessity to pass the result of the original function call to the test functions. Another reason for this implementation decision is that, since the function calls are in different processes, challenges would arise in comparing the outputs if the results are pointers,

which would point to memory in separate process spaces. The possible side effect of our implementation is that the original function call may alter the system state in such a way that the metamorphic property would not be expected to hold by the time the test function is called, possibly introducing false positives. In the experiments described below, none of the selected metamorphic properties fell into this trap, but further investigation needs to be performed to determine how often this problem may arise.

This limitation also affects the solution for handling non-deterministic functions, described above, in which a modification to the random number generator is required to force determinism across multiple calls. We note here this solution would not be necessary if the two function calls actually happen in parallel in two separate processes, one of which is the result of a "fork" call in the other. In such a case, if the code were using a random number generator whose internal state were the same in each process after the fork, then the next call to the generator would yield the same result in each process (this is the case in both C and Java when using the built-in random number functions). For the reasons described in the previous paragraph, though, a workaround was required.

Another limitation of the testing framework is that it uses function calls as the insertion points for metamorphic tests. In our investigation of the source code of some of the applications of interest, we noticed that some functions were quite long (over 200 lines) and that we were limited to how many and what sorts of metamorphic properties we could check. Smaller functions may have yielded more opportunities for metamorphic testing. Checking metamorphic properties at arbitrary points within a function would require more complex instrumentation and code re-writing, which may not be desirable or even possible in some cases.

4.4 Case Studies

To demonstrate the feasibility of Metamorphic Runtime Checking, we applied it to some open-source machine learning applications. The goal was to assess whether or not real defects could be revealed using the approach.

Our testing involved the Naive Bayes [88] implementations in both Weka 3.5.8 [194] and RapidMiner 4.1 [155]. Both Weka and RapidMiner provide Java implementations for numerous machine learning and data mining algorithms, and are popular tools for the development of Java machine learning applications. Naive Bayes is a probabilistic approach to the machine learning classification problem that assumes independence between the attributes, i.e., that the presence of a particular attribute in a class is unrelated to the presence of any other. The model is a formula that considers each attribute value and weighs it by its likelihood of correlating with the label.

4.4.1 Experimental Setup

For each of the implementations, we first determined the metamorphic properties of the entire application using the approach described in Section 2.4, and then identified various functions that reflected those same properties. We specified a total of 10 metamorphic properties for the two applications we investigated. Both implementations share the following properties:

1. Permuting the order of the examples in the training data should not affect the model

2. Permuting the order of the examples in the testing data should not affect their classification

3. If all attribute values in the training data are multiplied by a positive constant, the new model should be semantically equivalent to the original (meaning that, if the values in the testing data were multiplied by the same constant, the new model would

classify them with the same label as in the original model)

4. If a positive constant is added to all attribute values in the training data, the new model should be semantically equivalent to the original

Additionally, the Weka implementation provides an API for updating an existing model with a new example; if this is done, it should yield the same model created with training data originally containing that example. Also, for the RapidMiner implementation, a confidence value is reported when each example is classified. If an example that exists in the training data is classified, and the model is changed so that the example exists in the training data twice, the reported confidence should be half as much (since a lower confidence value means "more confident")

We then annotated the corresponding methods with specifications using the notation described above, and used Columbus to pre-process the source code to create the test methods. Last, we used data sets from the UC-Irvine Machine Learning Repository [138] (the same ones listed in Table 3.2 in Section 3.4.2) to perform testing, and looked for violations of properties, which would indicate that a defect had been found. 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 Section 4.4.3 below).

4.4.2 Findings

The Naive Bayes implementation in Weka provides 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 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 update the probability estimates, thus causing a difference compared to the model built using the entire data set. The Weka developers told us that this was by design, and not actually a defect in the code. However, because the behavior deviated from what the user of the implementation may have expected, we could say in this case that metamorphic testing was shown to be useful for validation (i.e., determining whether this implementation is right for the task at hand) more than verification [195].

We also 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 turned out to be a known defect in the version we tested, and was fixed in a later release.

4.4.3 Discussion

In practice, the use of Metamorphic Runtime Checking to specify metamorphic properties would seem to work best for methods that both take input and produce output, so that changes to the input of a function can produce an output that can be predicted and then analyzed easily. For instance, in the NaiveBayes implementation in both Weka and RapidMiner, the Java classes contained methods that took a single example as input and produced

a classification as output. However, in the Weka implementation's training phase, there was no single method that took the training data as input *and* produced a model as output. Rather, the training data was input as a parameter but the model was represented by one or more member variables in the class, modified by a side effect. Thus, to compare the models after changing the input, a call to a separate method was required, and there was no way to call all of the necessary methods in one single-line specification of the metamorphic property using our notation.

However, to work around this restriction, we found that it was rather straightforward to write a new test function that would conduct the metamorphic test on its own: it would take as its arguments the example to be classified and the result from the original method call, perform the metamorphic transformation, call the necessary method(s), and then compare the results. The metamorphic property that we specified for the method would simply be that this new test function should return true, similar to the example shown in Figure 4.8.

Overall, our case study demonstrated that Metamorphic Runtime Checking is a feasible approach for detecting defects in applications without test oracles, and that the Columbus framework is flexible enough to allow for the specification of metamorphic properties beyond simple identity functions. Note that, in this study, the properties used by Metamorphic Runtime Checking mirrored those of the application as a whole. This is not a limitation of the approach, though: it is possible that individual functions may have properties that the entire application does not, and those properties may provide extra fault-revealing power, as shown in the next section.

4.5 Empirical Studies

To measure the effectiveness of Metamorphic Runtime Checking, we conducted experiments on the same applications used in the studies presented in the previous chapter (Section 3.4), and compare the results against using metamorphic properties at the system level alone.

Our intuition is that Metamorphic Runtime Checking will only be effective for detecting defects in functions that actually have metamorphic properties, but that it will be more effective than system-level metamorphic testing in those cases.

The goal of these studies is to answer the following research questions:

1. Can Metamorphic Runtime Checking reveal defects not found by using system-level metamorphic properties alone?

2. Is Metamorphic Runtime Checking more effective at detecting defects in functions for which metamorphic properties have been identified?

4.5.1 Study #1: Machine Learning Applications

In this experiment, we revisit the same four machine learning applications used in the first study presented in the previous chapter (Section 3.4.2).

The four applications, described previously in Section 2.5, are as follows:

1. C4.5 release 8 [154], which uses a decision tree to perform classification and is written in C

2. The machine learning ranking algorithm MartiRank [70], also written in C, developed by researchers at Columbia University's Center for Computational Learning Systems

3. Support Vector Machines (SVM) [183], a linear classifier as implemented in the popular Weka [194] 3.5.8 open-source toolkit for machine learning in Java

4. The anomaly-based intrusion detection system PAYL [186], implemented in Java by researchers in Columbia University's Intrusion Detection System Lab

Experimental Setup

As in Section 3.4.2, we used mutation testing to systematically insert defects into the code and see how many were detected. We used the same mutated versions of the applications

as in the previous experiment. For C4.5, there were 28 versions, each with one mutant; for MartiRank, there were 69; for SVM, 85; and for PAYL, 40 versions.

Next we investigated the source code, determined the metamorphic properties at the function-level, and verified that they would also hold in the "gold standard". Note that we are not the developers of any of the four applications used in the experiment, so we did not have particularly intimate knowledge of the code (we did, admittedly, have direct access to the developers of MartiRank and PAYL). Even without being very familiar with the code, though, when it came to identifying metamorphic properties for use in the experiment, we were able to use the guidelines described in Section 2.4. For C4.5, MartiRank, and SVM, we identified four function-level metamorphic properties, and two properties for PAYL. The function-level metamorphic properties are listed in Table 4.3; note that these properties are not necessarily the same as the ones for the entire system, but rather are separate properties that apply to the particular selected functions.

For each mutated variant, we used the Columbus framework with the same program inputs as described in the experiment in Section 3.4.2 to determine whether Metamorphic Runtime Checking would reveal the defect. The goal of the experiment is to determine how many of the defects are detected by Metamorphic Runtime Checking, and to show that an approach based on checking the metamorphic properties of individual functions can find defects not revealed by checking system-level properties alone.

Results

Table 4.4 shows the results of the experiment for the four machine learning applications. The third column represents the number of mutants killed in the study presented in Section 3.4.2 using system-level metamorphic testing, and the last column shows the number killed by Metamorphic Runtime Checking.

Table 4.5 shows the results with the mutants grouped by (a) those that were killed by both approaches, (b) those that were killed only by the system-level metamorphic properties,

ID	App.	Function	Function Description	Metamorphic Property
C1	C4.5	FormTree	Creates decision tree	Permuting the order of the examples in the training data should not affect the tree
C2	C4.5	FormTree	Creates decision tree	Multiplying each element in the training data by a constant should yield the same tree, but with the values at decision points also increased
C3	C4.5	FormTree	Creates decision tree	Negating each element in the training data should yield the same tree, but with the values at decision points negated and the comparison operators reversed
C4	C4.5	Classify	Classifies example	Multiplying the values in the example should yield the same classification if the values at decision points are also similarly increased
M1	MartiRank	pauc	Computes "quality" [76] of a ranking	\result = 1 - reverse ranking
M2	MartiRank	sort_examples	Sorts set of examples based on given comparison function	Permuting the order of the elements and negating them returns the same result, but with the elements in the reverse order
M3	MartiRank	sort_examples	Sorts set of examples based on given comparison function	Multiplying the elements by a constant returns the same result
M4	MartiRank	insert_score	Inserts a value into an array used to hold top N scores	Calling the function a second time with the same value to be inserted should not affect the array of scores
P1	PAYL	computeTCP-LenProb	Computes probability of different lengths of TCP packets	Changing the byte values and permuting their order does not change the results
P2	PAYL	testTCPModel	Returns distance between an instance and corresponding "normal" instance in the model	Permuting the order of the elements in the model and multiplying all values by a constant c affects the result by a factor of c
S1	SVM	distribution-ForInstance	Estimates class probabilities for given instance	Adding a class that is not strongly associated with the instance should not affect the classification
S2	SVM	buildClassifier	Creates model from set of instances (training data)	Randomly permuting the order of the instances should yield the same model
S3	SVM	buildClassifier	Creates model from set of instances (training data)	Adding a constant to the values of the instances should yield the same model but with all values increased
S4	SVM	SVMOutput	Computes output (distance from hyperplane) for given instance	If all instances in model have values negated, and given instance does as well, output should stay the same

Table 4.3: Function-level metamorphic properties used in Study #1

Application	Total Mutants	Mutants killed with system-level properties	Mutants killed with Metamorphic Runtime Checking
C4.5	28	26 (93%)	10 (38%)
MartiRank	69	69 (100%)	39 (57%)
SVM	85	83 (97%)	60 (71%)
PAYL	40	2 (5%)	29 (73%)
Total	**222**	**180 (81%)**	**138 (62%)**

Table 4.4: Distinct defects found in Study #1

(c) those that were killed only by Metamorphic Runtime Checking, and (d) those that were not killed by either one.

Application	Total Mutants	Mutants killed by BOTH approaches	Mutants killed ONLY with system-level properties	Mutants killed ONLY with Metamorphic Runtime Checking	Mutants not killed
C4.5	28	10	16	0	2
MartiRank	69	39	30	0	0
SVM	85	59	24	1	1
PAYL	40	2	0	27	11
Total	**222**	**110**	**70**	**28**	**15**

Table 4.5: Defects found in Study #1, grouped according to the techniques that discovered them

Discussion

The fact that Metamorphic Runtime Checking was not as effective overall (Table 4.4) as testing based on system-level properties is not terribly surprising, since not all of the functions that contained mutants also had metamorphic properties. If we consider mutants only in those functions for which metamorphic properties were identified (as listed in Table 4.3), we can see that Metamorphic Runtime Checking is actually more effective when considering all four applications, as shown in Table 4.6.

The application for which Metamorphic Runtime Checking was more effective was PAYL. As shown in Table 4.5, Metamorphic Runtime Checking killed 27 mutants that metamorphic testing based on system-level properties did not. The improvement is admittedly low-hanging fruit, since the system-level approach had very little success to begin with. In particular, only very basic properties could be used: permuting the ordering of the input

Application	Total Mutants	Mutants killed with system-level properties	Mutants killed with Metamorphic Runtime Checking
C4.5	11	10 (91%)	10 (91%)
MartiRank	40	40 (100%)	38 (95%)
SVM	60	59 (98%)	60 (100%)
PAYL	29	2 (5%)	29 (100%)
Total	**140**	**111 (79%)**	**137 (98%)**

Table 4.6: Distinct defects found in Study #1, considering only functions identified to have metamorphic properties

data (which were network packets), and permuting the ordering of the bytes within those packet payloads. It was not possible to conduct system-level metamorphic tests based on modifying the values of the bytes inside the payloads (say, increasing them), not because of a limitation of the approach, but because the application itself only allowed for particular syntactically and semantically valid inputs that reflected what it considered to be "real" network traffic. However, once we could use Metamorphic Runtime Checking to put the metamorphic tests "inside" the application, we were able to circumvent such restrictions and perform tests using properties of the functions that involved changing the byte values. Thus, we were able to create more complex metamorphic tests that revealed additional defects.

An interesting revelation from Tables 4.5 and 4.6 is that Metamorphic Runtime Checking was able to find a defect that was not found by system-level metamorphic testing in SVM. This is perhaps the most important finding of the experiment. The defect was in the "distributionForInstance" function, which, for each classification (label) in the training data, uses the model to estimate the probability that a given example from the testing data belongs to that class. The metamorphic property (S1 in Table 4.3) is that adding a class to the model that is not strongly associated with the instance (i.e., for which there is almost no probability that the instance will be classified as such) should not affect the classification. This can be accomplished by adding a new example to the model with attribute values that are very, very far away (in K-dimensional space) from the example being classified.

Figure 4.10 shows the code snippet for which Metamorphic Runtime Checking was able to reveal the defect. On line 1346, an array called `result` is created to hold the probabilities that the given instance `inst` will belong to each class, i.e., `result[i]` represents the probability that `inst` will be classified with label `i`. On lines 1347 and 1348, the code loops so that each class is compared to each other class. The call to `SVMOutput` on line 1351 determines which of the two classes the instance is more likely to belong to. Lines 1352-1355 keep a running count of which class "wins" the comparison as a result of `SVMOutput`, the idea being that the one that "wins" the most will have the highest value in the `result` array, and other classes will have smaller values accordingly.

```
1346  double[] result = new double[inst.numClasses()];
1347  for (int i = 0; i < inst.numClasses(); i++) {
1348    for (int j = i + 1; j < inst.numClasses(); j++) {
1349      ...
1350      ...
1351      double output = m_classifiers[i][j].SVMOutput(-1, inst);
1352      if (output > 0) {
1353        result[j] += 1;
1354      } else {
1355        result[i] += 1;
1356      }
1357    }
1358  }
```

Figure 4.10: Snippet of code from SVM source used to determine class probabilities for a given instance.

Now consider a defect on line 1352 in which the comparison operation is switched from greater than to less than. Now the `result` array tracks the number of times each class "loses". Given the system-level metamorphic properties used in the first experiment, the results will never change, since adding, multiplying, negating, or permuting the examples does not affect "winning" and "losing"; even though the final output is incorrect, the application of the metamorphic property will not change the result of determining the "winner", so the property is never violated, and the defect is not discovered.

On the other hand, the defect *is* discovered by Metamorphic Runtime Checking, because

of the property (S1) that is being checked. The metamorphic property of this function is expected to hold because, when extra classes are added, the relative values in `result` never change if those new classes always "lose", since all the other classes will simply get that many more "wins". Given the mutant on line 1352, though, now that `result` is tracking "losses" instead of "wins", the newly-added class will have the highest value in `result` since it always "loses". Thus, because it has the highest value, the classification of the instance will change, in violation of the metamorphic property, thus revealing the defect.

Note also that system-level metamorphic testing did not detect this defect because there is no system-level metamorphic property that corresponds to function-level property S1. It is true that a similar property exists: adding a new label to the training data should not affect the classification of other examples if there is no strong association between any example and the new class (i.e., it is very unlikely that any example will be classified with the given label). However, even though the defect on line 1352 causes the output to be incorrect, this system-level property still holds, because the property calls for the introduction of a new class that does not affect *any* of the examples, and none of the classifications change. When we use Metamorphic Runtime Checking, though, we get more fine-grained detail and we are able to introduce a new class that we know will not affect a *single* example, but as described above, the metamorphic property is violated because of the defect.

As for MartiRank, we noticed that only 40 of the 69 mutants (58%) from the original study were in functions for which we could identify metamorphic properties, which partially explains the poor results for Metamorphic Runtime Checking of MartiRank in Table 4.4. We discovered that almost all of the remaining 29 mutants were in the "main" function, for which we did not identify metamorphic properties, but constitutes over 40% of the MartiRank code (approximately 500 lines out of a total of 1200). Since the "main" function essentially represents the program as a whole, it follows that system-level metamorphic testing would be more effective at finding defects in it, of course. For C4.5, only 11 of the 28 mutants (39%) were in the functions that had metamorphic properties; the explanation is

that we only identified two functions with metamorphic properties, yet defects resided in many others.

Analysis of Metamorphic Runtime Checking

Here we analyze the effectiveness of the different properties used in Metamorphic Runtime Checking. Tables 4.7, 4.8, 4.9, and 4.10 show the number of mutants killed (in all functions) by the metamorphic properties for C4.5, MartiRank, SVM, and PAYL, respectively, as identified in Table 4.3.

Mutation	**Mutants**	**C1**	**C2**	**C3**	**C4**	**Total**
Comparison operators	8	2	0	2	0	2 (25%)
Math operators	15	5	0	7	0	7 (47%)
Off-by-one	5	0	0	1	0	1 (20%)
Total	**28**	**7**	**0**	**10**	**0**	**10 (38%)**

Table 4.7: Results of Mutation Testing for C4.5 using Metamorphic Runtime Checking

Mutation	**Mutants**	**M1**	**M2**	**M3**	**M4**	**Total**
Comparison operators	20	10	16	0	4	18 (90%)
Math operators	23	11	11	0	5	11 (47%)
Off-by-one	26	10	2	0	7	10 (38%)
Total	**69**	**31**	**29**	**0**	**16**	**39 (59%)**

Table 4.8: Results of Mutation Testing for MartiRank using Metamorphic Runtime Checking

The metamorphic properties based on multiplication (C2, C4, and M3) all were unable to reveal any defects. This is consistent with the results of the first study using system-level properties, in which the multiplicative properties were consistently the poorest performers. The explanation is that for the operations that were changed by the mutations, they would still yield the same results because of the distributive properties of multiplication. As a

Mutation	Mutants	S1	S2	S3	S4	Total
Comparison operators	30	15	10	16	11	16 (53%)
Math operators	24	5	18	22	16	24 (100%)
Off-by-one	31	7	20	11	20	20 (65%)
Total	**85**	**27**	**48**	**49**	**47**	**60 (82%)**

Table 4.9: Results of Mutation Testing for SVM using Metamorphic Runtime Checking

Mutation	Mutants	P1	P2	Total
Comparison operators	15	7	3	7 (47%)
Math operators	7	2	4	4 (57%)
Off-by-one	18	14	16	18 (100%)
Total	**40**	**23**	**23**	**29 (73%)**

Table 4.10: Results of Mutation Testing for PAYL using Metamorphic Runtime Checking

very simple example, consider a function $f(x, y) = x + y$. We would expect it to have the metamorphic property $f(2x, 2y) = 2f(x, y)$. Now consider a mutation of this function in which the plus sign has been replaced with a minus sign: $f'(x, y) = x - y$. Although there is an defect in the code, clearly the metamorphic property $f'(2x, 2y) = 2f'(x, y)$ still holds; thus, the metamorphic property based on multiplication would not show a violation.

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

Many of the other results were consistent with the system-level metamorphic testing study, specifically that the property based on negation (C3) was most effective in C4.5

because of the impact the mutations had on the creation of the decision tree, and that properties based on permuting (such as S2) were effective at killing off-by-one errors in applications such as SVM that relied on mathematical calculations; the same was observed to be true in PAYL, in which the defects that were discovered were in functions that performed computations.

One of the most interesting findings can be seen when considering the results in Tables 4.4 and 4.6. For both MartiRank and C4.5, Metamorphic Runtime Checking found defects in functions that did not have metamorphic properties, but rather were in functions other than the ones in which the metamorphic properties were actually being checked. The defects actually existed outside those functions, but put the system into a state in which the metamorphic property of the function would be violated. For instance, the `pauc` function in MartiRank uses an array of numbers (which is part of the application state) and performs a calculation on them to determine the "quality" [76] of the ranking, returning a normalized result (i.e., between 0 and 1). One of the metamorphic properties of that calculation (property M1) is that reversing the order of the values in the array should produce the "opposite" result, i.e., `pauc`(*A*) = 1 - `pauc`(*A'*) where *A'* is the array in which the values of *A* are in reverse order. However, a defect in a *separate* function that deals with how the array was populated caused this property to be violated because the data structure holding the array itself was in an invalid state, even though the code to perform the calculation was in fact correct (we know it was correct, of course, because we know where the defect was seeded in that case).

As a slight simplification, we can explain this as follows: the values in the array *A* were being stored in a doubly-linked list, so that MartiRank could calculate the "quality" of the list by looking at it forwards (ascending) and backwards (descending). An off-by-one mutation in the function that created the linked list caused some of the links to "previous" nodes to point to the wrong ones, as in Figure 4.11. In this case, traversing the linked list in the forward direction would give *ABCDE*, but backwards would give *EDBCA*.

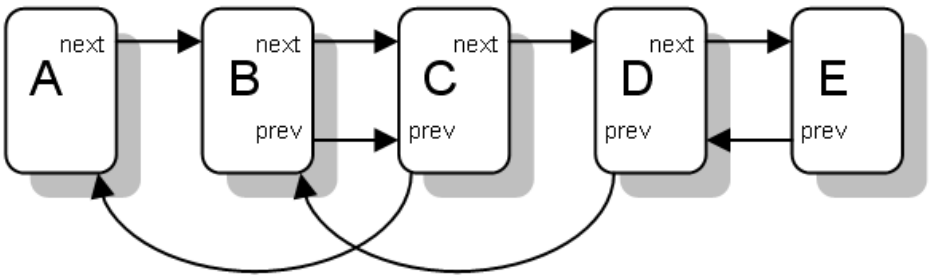

Figure 4.11: A doubly-linked list in which elements B, C, and D point to the wrong nodes. The defect that caused this error was detected using a metamorphic property of another function.

The metamorphic property that `pauc`(*A*) = 1 - `pauc`(*A'*) would only hold if *A'* were, in fact, the exact opposite ordering of *A*, but clearly in this case it is not. Note that in this case, Metamorphic Runtime Checking detected a defect caused by an invalid application state, and not a defect in the function under test itself (i.e., the one with the metamorphic property).

These results demonstrate the real power of Metamorphic Runtime Checking: without much knowledge of the details of the entire implementation, we were able to detect many of the defects by simply specifying the expected behavior of particular functions, *even though the defects were not always in those functions*; rather, those defects created violations of the metamorphic properties because they put the system into an invalid state. Although we have yet to demonstrate this quantitatively, alternative approaches to detecting such invalid states (such as checking data structure integrity [52] or algebraic specifications [143]) require more intimate familiarity with the source code, such as the details of pointer references or data structures, or dependencies between variables, as opposed to simply specifying how a function should behave when its inputs are modified, using the guidelines described previously to identify metamorphic properties.

4.5.2 Study #2: Applications in Other Domains

In this experiment, we investigated the same three applications as in the second study of the previous chapter (Section 3.4.3): JSim [193], a discrete event simulation tool implemented in Java by researchers at the Laboratory for Advanced Software Engineering Research at the University of Massachusetts-Amherst; Lucene [6], which is a text search engine library that is part of the Apache framework and is written in Java; and gaffitter [10], which uses a genetic algorithm implemented in C++ to solve the bin-packing problem.

Experimental Setup

For each of the three applications, the same mutated versions were used as in the study from Section 3.4.3. For JSim, there were six versions, each with one mutant; for Lucene, there were 15; and for gaffitter, 66.

For JSim, the metamorphic properties were identified in the Task class. In particular, we instrumented the `compareTo` method, which compares two objects in a semantically meaningful manner. It has the property that, for objects *A* and *B*, *A*.`compareTo`(*B*) should equal -1 * *B*.`compareTo`(*A*).

For Lucene, we identified metamorphic properties in methods in the same two classes used in the system-level metamorphic testing experiment. In particular, all of the usable mutants were in the `idf` method, which calculates the inverse document frequency (IDF) of a given search result. The IDF indicates how important a particular word is to a document in the overall group [163], and is defined as $\texttt{log}_{10}\texttt{(numDocs/(docFreq+1)) + 1}$, where `numDocs` is the number of documents in the corpus and `docFreq` is the frequency with which the word appears.

Given this definition, this function has the metamorphic property that if the `docFreq` value is multiplied by 10 and then increased by 9, the total value should decrease by approximately 1 (within the tolerance of the floating point calculation), since:

$\texttt{log}_{10}\texttt{(numDocs/((10*docFreq+9)+1))}$

= $\log_{10}$(numDocs/(10docFreq+10))

= $\log_{10}$(numDocs/10(docFreq+1))

= $\log_{10}$((1/10)(numDocs/(docFreq+1)))

= $\log_{10}$(numDocs/(docFreq+1)) + $\log_{10}$(1/10)

= $\log_{10}$(numDocs/(docFreq+1)) - 1 .

Additionally, if numDocs is multiplied by 10, the total value should increase by approximately 1, given that:

$\log_{10}$(10*numDocs/(docFreq+1))

= $\log_{10}$((10)(numDocs/(docFreq+1)))

= $\log_{10}$(numDocs/(docFreq+1)) + $\log_{10}$(10)

= $\log_{10}$(numDocs/(docFreq+1)) + 1 .

Table 4.11 lists the metamorphic properties used for gaffitter. These properties are discussed further in the "Discussion and Analysis" section below.

ID	Function	Description	Property
G1	Crossover	Merges two candidate solutions to create a new one	If the order of the arguments is reversed, the new child should contain all elements that do not appear in the original
G2	Fitness	Determines the fitness of a candidate solution (how close it is the the optimum)	Permuting the order of the elements in the candidate solution should not affect the result
G3	Fitness	Determines the fitness of a candidate solution (how close it is the the optimum)	Multiplying each element in the candidate solution by a constant should not affect the result is the target is multiplied by the same constant
G4	Generation	Determines which candidate solutions survive into the next generation	Permuting the ordering of the candidate solutions should not affect the result

Table 4.11: Metamorphic properties of gaffitter used in Study #2.

Results

Table 4.12 shows the number of defects detected by each approach for the three applications investigated in this study. Table 4.13 breaks down the results according to the testing

techniques that detected the defects.

App.	Total Mutants	Mutants killed with system-level properties	Mutants killed with Metamorphic Runtime Checking
JSim	6	6 (100%)	2 (33.3%)
Lucene	15	11 (73.3%)	9 (60%)
gaffitter	66	22 (33.3%)	34 (51.5%)

Table 4.12: Distinct defects detected in Study #2.

Application	Total Mutants	Mutants killed by BOTH approaches	Mutants killed ONLY with system-level properties	Mutants killed ONLY with Metamorphic Runtime Checking	Mutants not killed
JSim	6	2	4	0	0
Lucene	15	6	5	3	1
gaffitter	66	20	2	14	30
Total	**87**	**28**	**11**	**17**	**31**

Table 4.13: Distinct defects detected in Study #2, grouped by the testing techniques that discovered them.

Discussion and Analysis

As in the first study, it is perhaps not surprising that Metamorphic Runtime Checking was not as effective as metamorphic testing based on system-level properties for JSim and Lucene, since we were only able to identify a small number of metamorphic properties at the function level, and not all functions that had defects also had metamorphic properties.

Of the mutants not killed by Metamorphic Runtime Checking, we noticed that four of them were in the constructor of the JSim Task class, and another was in the constructor of the Lucene TermScorer class. Given that constructors typically don't produce "output" per se, this raised challenges in identifying metamorphic properties, especially given that both constructors take only a small number of arguments but rely on other methods (sometimes from other classes) to populate the attribute values. We were thus unable to identify any metamorphic properties of these constructors, though this is a limitation of our own understanding of the source code, and not of the approach or testing framework themselves. Of course, a defect in the constructor could conceivably be revealed by the violation of a metamorphic property of *another* method, though this did not happen in our case.

However, for Lucene, we see from Table 4.13 that Metamorphic Runtime Checking was able to kill three of the mutants not killed by system-level metamorphic testing. Perhaps not coincidentally, those three were all in the `idf` function of the DefaultSimilarity class, for which function-level metamorphic properties were identified.

In this particular case, the system-level metamorphic properties only considered how the results of different calculations compared to each other, but not their actual values. Consider a simple off-by-one mutant such that the `idf` method returns a value that is one greater than it should be. At the system level, the properties that were specified could not access the value returned by `idf`, since the results of the individual calculations were not directly reflected in the program output. For instance, for the metamorphic property that the search results for "romeo OR juliet" should be the same as "juliet OR romeo", it is clear that the `idf` value for each document will still be the same in each case, even though there is an error in the computation, so the metamorphic property would still hold. However, when we used Metamorphic Runtime Checking, we could see that there was a violation in the property of `idf`, revealing the defect.

For gaffitter, one of the reasons why Metamorphic Runtime Checking was more effective than system-level metamorphic testing is that we were able to identify more properties. This is similar to what we found with PAYL in the first study: the restricted input domains of some applications make it difficult to specify metamorphic properties, but this can be circumvented once the tests are "inside" the code.

Table 4.14 shows the effectiveness of the different metamorphic properties used for testing gaffitter. Not surprisingly, the properties based on permuting the input (G2 and G4) were only effective for killing mutants related to off-by-one errors.

As mentioned in the system-level metamorphic testing study, many of the mutants related to calculations in gaffitter had only slight influence on the final output, and were not revealed by system-level metamorphic properties. Consider the pseudo-code in Figure 4.12, which summarizes lines 212-226 in GeneticAlgorithm.cc and is used to create a

Mutation	Mutants	G1	G2	G3	G4	Total
Comparison operators	15	4	0	2	0	4 (27%)
Math operators	19	6	0	7	0	12 (63%)
Off-by-one	32	10	6	0	5	18 (56%)
Total	**66**	**20**	**6**	**9**	**5**	**34 (52%)**

Table 4.14: Results of Mutation Testing for gaffitter using Metamorphic Runtime Checking

child candidate solution from two parent solutions, *CS1* and *CS2*. One of its metamorphic properties is that it is easy to calculate the expected result if the ordering of the arguments is switched, i.e., if we call "`crossover`(*CS2*, *CS1*)", assuming of course that the random number generation is deterministic, as explained above. For instance, suppose *CS1* = {1, 2, 3, 4} and *CS2* = {5, 6, 7, 8, 9} (keeping in mind that the number of elements in each need not be the same). If the crossover point were the second element of *CS1*, then the child should be {1, 2, 7, 8, 9}. If we reverse the ordering of the arguments so that we start with *CS2*, we should get {5, 6, 3, 4}. This is easy to predict, given the first result: it just contains the elements of *CS1* and *CS2* that were not contained in the original output.

An off-by-one mutant on line 4 may, however, affect the result and would violate the function's metamorphic property. For instance, if instead of taking elements up to `length`(*CS2*), the defect caused it only to take elements to `length`(*CS2*)-1, then the original output would be {1, 2, 7, 8} and the follow-up output (after applying the metamorphic transformation) would be {5, 6, 3}. It is clear that this would violate the property, since elements 4 and 9 would not be included. As argued in the previous chapter, though, the exclusion of one element from the child is unlikely to have much effect on the final overall result, which is why the defect was not found by the system-level property. However, when we consider the properties of the individual function, this defect is revealed.

As another example of a defect that Metamorphic Runtime Checking discovered but system-level metamorphic testing did not, consider the simple pseudo-code in Figure 4.13

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

Figure 4.12: Sample code from genetic algorithm to perform crossover between two candidate solutions

to calculate the "fitness" of a given candidate solution, i.e., how close to the optimal solution (target) a candidate comes. A metamorphic property of the function is that changing the ordering of the elements in the candidate solution should not affect the result, since it is merely taking a sum of all the elements.

If, for instance, there is an off-by-one error on line 3 (so that the last element is omitted from the calculation), then the metamorphic property will be violated since the return value will be different after the second function call. However, at the system level, such a defect is unlikely to be detected, since the metamorphic property simply stated that the quality of the solutions should be increasing with subsequent generations. Even though the *value* of the fitness is incorrect, it would still be increasing (unless the omitted element had a very large effect on the result, which is unlikely), and the property would not be violated.

```
1  fitness(CS, target) {
2     total = 0;
3     for (int i = 0; i < length(CS); i++
4        total += size(CS[i]);
5     if (total > target) return -1;
6     else return total/target;
7  }
```

Figure 4.13: Sample code from genetic algorithm to calculate quality of candidate solution

To summarize, Table 4.15 shows the results when only considering functions that were identified to have metamorphic properties. These results, as well as the findings for certain defects in Lucene and gaffitter, indicate that Metamorphic Runtime Checking is more

effective than testing based on system-level properties alone, particularly for defects in calculations that only have a subtle effect on the overall output of the program. Although these small deviations from the specification may not matter from a practical point of view, they do result in the violation of a sound property, and thus fit our definition of a "defect".

App.	Total Mutants	Mutants killed with system-level properties	Mutants killed with Metamorphic Runtime Checking
JSim	2	2 (100%)	2 (100%)
Lucene	9	6 (66.7%)	9 (100%)
gaffitter	38	9 (23.7%)	34 (89.4%)

Table 4.15: Defects detected in Study #2, considering only those functions for which metamorphic properties were identified.

4.5.3 Study #3: Non-Deterministic Applications

In the third study, we applied Metamorphic Runtime Checking to non-deterministic applications. The first is MartiRank, which, as noted previously, is non-deterministic in the ranking phase when there are missing or unknown attributes in the testing data set. The other application is the simulator JSim, which can be non-deterministic if the timing of different events is configured to be random over a specified interval.

Experimental Setup

As with the previous studies, we used the same mutated versions as in the system-level metamorphic testing experiment, described in Section 3.4.4. For MartiRank, there were 59 versions, each with one mutant. For JSim, there were 19 versions with mutants related to event timing.

In this study, we used the same function-level metamorphic properties for MartiRank as in Study #1 (Section 4.5.1), summarized again here in Table 4.16. Note that the non-determinism in MartiRank is found in the `sort_examples` method, for which two metamorphic properties (M2 and M3) are identified. In order to use Metamorphic Runtime

ID	App.	Function	Function Description	Metamorphic Property
M1	MartiRank	pauc	Computes "quality" [76] of a ranking	\result = 1 - reverse ranking
M2	MartiRank	sort_examples	Sorts set of examples based on given comparison function	Permuting the order of the elements and negating them returns the same result, but with the elements in the reverse order
M3	MartiRank	sort_examples	Sorts set of examples based on given comparison function	Multiplying the elements by a constant returns the same result
M4	MartiRank	insert_score	Inserts a value into an array used to hold top N scores	Calling the function a second time with the same value to be inserted should not affect the array of scores

Table 4.16: Metamorphic properties of MartiRank used in Study #3

Checking with a non-deterministic function, we needed to "force" determinism by using a special random number generator, as described previously, that would ensure that the same random number would be returned in the test process.

For JSim, the mutants related to non-deterministic event timing were all in the `get` methods of the LinearRangeDuration and TriangleRangeDuration classes, which returns a random integer in the range [*min*, *max*]. These methods have two metamorphic properties: if `get`() returns *k* for range [*min*, *max*], then it should return *k*+10 for range [*min*+10, *max*+10], and 10*k* for range [10*min*, 10*max*]. As we did for MartiRank, in this case we needed to "force" determinism in the function in the metamorphic tests.

Results

Table 4.17 shows the number of defects discovered by the two testing approaches for MartiRank and JSim.

App.	Total Mutants	Mutants killed with system-level properties	Mutants killed with Metamorphic Runtime Checking
MartiRank	59	40 (67.7%)	34 (57.6%)
JSim	19	19 (100%)	19 (100%)

Table 4.17: Distinct defects detected in Study #3.

Discussion and Analysis

As in the previous studies, system-level metamorphic testing is generally more effective than Metamorphic Runtime Checking, since the latter is limited by the number of functions that are instrumented, and where the defects happen to reside. However, further analysis indicates that under certain circumstances, Metamorphic Runtime Checking can find defects that are not otherwise revealed.

Table 4.18 shows the results for MartiRank, broken down by the functions containing the mutations. The first row of data shows the results for the `pauc` and `insert_score` functions, which are deterministic and have metamorphic properties M1 and M4; `sort_examples` is non-deterministic, and has metamorphic properties M2 and M3; the other functions are all deterministic, but were not identified to have metamorphic properties in this study.

Function(s)	**Total Mutants**	**Mutants killed with system-level properties**	**Mutants killed with Metamorphic Runtime Checking**
`pauc` and `insert_score`	20	16 (80%)	16 (80%)
`sort_examples`	18	10 (56%)	18 (100%)
other	21	14 (67%)	0 (0%)
Total	**59**	**40 (68%)**	**34 (58%)**

Table 4.18: MartiRank defects detected in Study #3.

As in the previous studies, Metamorphic Runtime Checking is shown to be at least as effective as system-level metamorphic testing when only considering defects in functions for which metamorphic properties are being checked (`pauc` and `insert_score`). In this case, we also see that for the non-deterministic function `sort_examples`, Metamorphic Runtime Checking was able to reveal a number of defects not found by metamorphic properties of the entire application.

The explanation is that the similarity metric calculated by Heuristic Metamorphic Testing (introduced in Section 3.3) was too forgiving to be effective at detecting some of the mutants. For instance, in some cases, Heuristic Metamorphic Testing determined that the similarity of the results (using the normalized equivalence measure) was around 80%;

after applying the metamorphic transformation, the similarity was still around 80% (i.e., not showing a statistically significant difference), and the property was considered to be upheld. Now, 80% is not particularly "similar", but because Heuristic Metamorphic Testing is essentially a pseudo-oracle approach, in that one cannot know what is correct but only checks that the results are as expected, this passed the test.

However, when we used Metamorphic Runtime Checking with the non-deterministic function, the property was violated and the defect was revealed. This is not to say that Heuristic Metamorphic Testing (as well as its cousin, statistical metamorphic testing) is not without merit, since sometimes it may not be possible to identify metamorphic properties of a non-deterministic function, but this shows that Metamorphic Runtime Checking can be more effective than system-level metamorphic testing in some cases.

Upon analyzing the MartiRank results, it perhaps comes as no surprise that property M2 (based on permuting) killed all of the mutants in the non-deterministic `sort_examples` function, whereas M3 (based on multiplying) killed none, since M3 did not detect any defects in Study #1, either. Given the types of mutants that were placed into the code, changing their values are not likely to reveal a violation of the metamorphic property (see, for instance, the bubble sort example from Section 3.4.2).

The effectiveness of Metamorphic Runtime Checking for detecting the defects related to non-deterministic event timing in JSim is to be expected, considering that all of the mutants were in the methods that we were checking. We note here that because *all* invocations of the methods violated the metamorphic property, a tester would have been alerted to this defect right away. On the other hand, with an approach like statistical metamorphic testing, which requires the program to run to completion numerous times, the tester would have to wait far longer to find out that the implementation was incorrect.

To summarize, this study demonstrates that Metamorphic Runtime Checking is effective at finding defects in non-deterministic applications, as it can use metamorphic properties to reveal errors in functions regardless of whether they are deterministic. Although statistical

and heuristic metamorphic testing approaches were shown to be effective in the study presented in Section 3.4.4, we have demonstrated here that Metamorphic Runtime Checking can find defects not found by using system-level properties.

4.5.4 Summary

Here we revisit the research questions that this study sought to answer:

1. Can Metamorphic Runtime Checking reveal defects not found by using system-level metamorphic properties alone?

2. Is Metamorphic Runtime Checking more effective at detecting defects in functions for which metamorphic properties have been identified?

For question #1, the answer is "yes". Of the 109 mutants not killed by using system-level metamorphic properties in the experiments in Section 3.4, 45 were detected by Metamorphic Runtime Checking. For question #2, when considering only the 226 mutations in functions for which metamorphic properties had been identified, system-level metamorphic testing killed 157 (69.4%), whereas Metamorphic Runtime Checking killed 219 (96.9%).

4.6 Effect on Testing Time

4.6.1 Developer Effort

The effectiveness of Metamorphic Runtime Checking heavily depends on the metamorphic properties that are specified by the tester or developer. The question often arises, "how long does it take to find and specify these properties?" We have not yet conducted empirical studies to measure this, but anecdotally we noted that, for each application used in the experiments in Section 4.5, it only took slightly more than an hour of analysis to identify the functions' metamorphic properties, primarily based on the guidelines suggested in

Section 2.4. Note that we are not the developers of any of the seven applications, nor had we performed much code analysis on the programs before using them in the studies. It follows that the software architect or developer, who would have more knowledge of the application than we did, would presumably be more efficient at identifying metamorphic properties.

Perhaps more formally, Hu et al. [85] conducted a study that measured the developer effort for deriving metamorphic properties. The subjects of the study were 38 graduate students who had no prior knowledge of metamorphic testing, nor of the applications to which they would apply the technique. After approximately three hours of training on the basics of metamorphic testing and some background about the three applications that they would test, the subjects were asked to identify metamorphic properties. On average, for each application the subjects were able to identify around four function-level metamorphic properties in a span of a little more than three hours; their study showed that these metamorphic properties were actually more effective at revealing defects than program invariants identified by the same subjects after a similar amount of training (see Section 6.2 in Related Work for more details). Thus, it can be concluded that with only about six hours of effort, a developer with no knowledge of metamorphic testing - or even much knowledge of the application under test - can identify metamorphic properties and effectively use an approach such as Metamorphic Runtime Checking.

4.6.2 Performance Overhead

Although Metamorphic Runtime Checking is able to detect defects not found by metamorphic testing based on system-level properties alone, this runtime checking of the properties comes at a cost, particularly if the tests are run frequently. In system-level metamorphic testing, the program needs to be run one more time with the transformed input, and then each metamorphic property is checked exactly once (just at the end of the program execution); as described in Section 3.2.7, when using the Amsterdam framework to automate

metamorphic testing, this overhead may be only a few hundred milliseconds if the invocations are run in parallel. In Metamorphic Runtime Checking, however, each property can be checked numerous times, depending on the number of times each function is called, and the overhead can grow to be much higher.

During the empirical studies discussed in Section 4.5, we measured the impact of the Columbus framework on the time it took to conduct testing. Tests were conducted on a server with a quad-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. Tables 4.19, 4.20, 4.21, and 4.22 show the results for C4.5, MartiRank, SVM, and PAYL, respectively, using the different data sets from the empirical studies and averaged over 10 runs.

Data Set	**Uninstrumented (ms)**	**Instrumented (ms)**	**Tests Executed**	**Time/Test (ms)**
glass	6	60	326	0.2
golf	2	8	28	0.2
heart	25	120	606	0.2
hepatitis	6	59	310	0.2
iris	3	53	300	0.2
wine	8	67	356	0.2
Total	**50**	**367**	**1926**	**0.2**

Table 4.19: Results of Metamorphic Runtime Checking performance tests for C4.5.

Data Set	**Uninstrumented (sec)**	**Instrumented (sec)**	**Tests Executed**	**Time/Test (ms)**
data1	10.8	24.0	26792	0.5
data2	24.3	38.8	26759	0.5
data3	24.3	37.8	26364	0.5
data4	11.0	23.1	26634	0.4
data5	0.3	2.0	10947	0.2
data6	0.6	4.7	26641	0.2
data7	0.7	4.8	26980	0.2
data8	0.7	4.7	26620	0.2
data9	0.1	5.5	45145	0.2
data10	22.2	58.7	53669	0.7
Total	**95.0**	**223.1**	**296551**	**0.4**

Table 4.20: Results of Metamorphic Runtime Checking performance tests for MartiRank.

The variations in the results come from the different numbers of functions that were instrumented in each application, the number of times those functions are called for each

Data Set	Uninstrumented (sec)	Instrumented (sec)	Tests Executed	Time/Test (ms)
glass	0.5	29.1	8492	3.3
golf	0.5	1.9	445	3.1
heart	3.1	216.2	59870	3.6
hepatitis	0.6	23.9	6668	3.6
iris	1.0	46.0	13694	3.4
wine	1.0	56.7	15489	3.6
Total	**6.7**	**373.8**	**104658**	**3.5**

Table 4.21: Results of Metamorphic Runtime Checking performance tests for SVM.

Data Set	Uninstrumented (sec)	Instrumented (sec)	Tests Executed	Time/Test (ms)
training	0.9	7.3	2300	2.7
testing	0.5	1.3	299	2.7
Total	**1.4**	**8.6**	**2599**	**2.7**

Table 4.22: Results of Metamorphic Runtime Checking performance tests for PAYL.

data set, and the implementation language. The time per test for SVM and PAYL is higher because the overhead is greater for Java applications: since Java does not have any "fork" utility, it needed to be implemented via a Java Native Interface call, which added extra overhead.

On average, the performance overhead for the Java applications was around 3.5ms per test; for C, it was only 0.4ms per test. This cost is mostly attributed to the time it takes to create the sandbox and fork the test process. Other overhead comes from context switching between the original process and the test processes in the cases where there are more tests than CPUs/cores.

This impact can certainly be substantial from a percentage overhead point of view if many tests are run in a short-lived program. For instance, Table 4.19 shows that the overhead is on the order of 10x for C4.5, even though in absolute terms it is just a few milliseconds. However, we point out that, for most the programs we investigated in our study, the overhead was typically less than a few minutes, which we consider a small price to pay for detecting that the output of the program was incorrect.

4.7 Summary

In this chapter, we have introduced *Metamorphic Runtime Checking*, a new technique for testing applications without test oracles, based on checking the metamorphic properties of individual functions, rather than just those of the entire system, as the application is running. Metamorphic testing of individual functions is carried out at the point when the function is called in the running program. If the metamorphic property is violated, then a defect has been revealed. This work goes beyond applying a system testing approach to individual functions: rather, we use properties of the functions to conduct testing of the overall application.

In describing Metamorphic Runtime Checking, we have also detailed an implementation framework called *Columbus*, and presented a notation for specifying metamorphic properties within the source code, so that they can be checked at runtime.

Our case study demonstrated the feasibility of the approach, and the results of our three empirical studies prove the hypothesis (stated in Section 1.6) that "an approach that conducts function-level metamorphic testing in the context of a running application will reveal defects not found by system-level metamorphic testing". Because we showed in the previous chapter that system-level metamorphic testing is more effective than other techniques at detecting defects in applications in the domains of interest, it is clear that a combined approach that also incorporates Metamorphic Runtime Checking would advance the state of the art even further.

In the next chapter, we generalize the Metamorphic Runtime Checking approach and describe how it can be used to check any properties (metamorphic or otherwise) in various types of applications, and consider the implications of conducting such testing of applications as they execute in the deployment environment.

Chapter 5

In Vivo Testing

During the development and assessment of the Metamorphic Runtime Checking approach, a few interesting research questions arose: Would the approach be effective in detecting defects in other types of applications, specifically those that *do* have test oracles? Can other properties be checked besides metamorphic properties? How feasible is it to continue the checking of such properties while the software runs in the deployment environment?

This chapter generalizes the Metamorphic Runtime Checking approach into a software testing technique whereby *any* properties of individual functions (not just metamorphic properties) can be checked while the application executes, even if the checking of the property has side effects that would alter the system state. Because the Metamorphic Runtime Checking approach essentially associates a test function with the original source code, there is no need for that test function to *only* execute metamorphic tests. It could, for instance, execute unit tests, integration tests, or conduct any other types of property checking.

In this new testing methodology, called *In Vivo Testing*, tests are continuously executed in the deployment environment, from within the running application, without altering the system's state from the end users' point of view. The approach can be used for detecting concurrency, security, or robustness issues, as well as defects that may not have appeared in

a testing lab (the "in vitro" environment). In Vivo Testing can use existing test code, or take advantage of *In Vivo tests* that exercise parts of the application as the system is running, no matter what its current state. These tests improve on traditional unit or integration tests by foregoing the assumption of a clean state created by a test harness, and focusing on aspects of the program that should hold true regardless of what state the system is in. These tests execute within the current state of the program without affecting or altering that state, as potentially visible to users.

In this chapter, we discuss our In Vivo Testing implementation framework and demonstrate that such an approach is feasible for testing an application even as it executes in the field. If the performance overhead can be kept at a minimum, and side effects from the tests can successfully be hidden from the user, then the tests can execute in the deployment environment and thus be carried out in a variety of configurations and/or application states that may not have been tested prior to release.

The rest of this chapter is organized as follows: Section 5.1 introduces In Vivo Testing and motivates the need for a deployment environment testing approach. Sections 5.2 and 5.3 describe the model and architecture, respectively, of the Invite implementation framework. Section 5.4 discusses the results of case studies that demonstrate the feasibility of the approach. Section 5.5 investigates the performance impact of In Vivo Testing, and Section 5.6 explains how the performance overhead can be reduced by running tests more efficiently.

5.1 Approach

Thorough testing of a software product is unquestionably a crucial part of the development process, but the ability to faithfully detect all defects in an application is severely hampered by numerous factors. A 2008 report [175] indicated that 40% of IT companies consider insufficient pre-release testing to be a major cause of later production problems, and the

problem only worsens as changes are rolled out into production without being thoroughly tested. Furthermore, it is possible that the test code itself may have flaws in it, too, perhaps because of oversights or incorrect assumptions made by the authors.

A key issue is that, for large, complex software systems, it is typically impossible in terms of time and cost to reliably test all configuration options before releasing the product into the field. For instance, Microsoft Internet Explorer has over 19 trillion possible combinations of configuration settings [46]. Even given infinite time and resources to test an application and all its configurations, once a product is released, the other software packages on which it depends (libraries, virtual machines, etc.) may also be updated; therefore, it would be impossible to test with these dependencies prior to the application's release, because they did not exist yet. A last 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 [110].

One proposed way of addressing this problem has been to continue testing the application in the field, after it has been deployed. The theory behind this notion of "perpetual testing" [148] is that, over time, defects will reveal themselves given that multiple instances of the same application may be run globally with different configurations, in different environments, under different patterns of usage, and in different system states.

The foundation of our approach to solving this problem is the fact that many (if not all) software products are released into deployment environments with latent defects still residing in them, as well as our claim that these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. The approach can be used to detect defects hidden by assumptions of a clean state in the tests, errors that occur in field configurations not tested before deployment, and problems caused by unexpected user actions that put the system in an unanticipated state; these flaws may also be due to corrupted states that could arise due to a security violation.

Our approach goes beyond passive application monitoring (e.g., [145]) in that it actively tests the application as it runs in the field.

In Vivo Testing is a methodology by which tests are executed continuously in the deployment environment, in the context of the running application, as opposed to a controlled or blank-slate environment. Crucial to the approach is the notion that the test must not alter the state of the application from the user's perspective. In a live system in the deployment environment, it is clearly undesirable to have a test alter 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 rest of this section motivates and describes the In Vivo Testing approach.

5.1.1 Conditions

In order for In Vivo Testing to be useful in practice, for a given test and a corresponding piece of software to be tested, three conditions must be met. First, the test must pass in the development environment, even though there are unknown defects in the software under test (if the test fails before deployment, then obviously In Vivo Testing is not necessary). Second, under certain potentially-unanticipated circumstances, the running application should give erroneous results or behavior in the deployment environment, i.e., have a defect. Last, for some process state or condition of use, the test must subsequently fail. If these conditions are met, it is possible for In Vivo Testing to detect that there is a defect. The defect may be one in the application code, or in the test code, or both.

5.1.2 In Vivo Tests

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 [121]), developers may find it desirable to create special In Vivo tests that are specifically designed to take advantage of the approach. These tests ensure that properties

of the function (or of subsystems, or even the entire application) hold true no matter what the application's state is. In the simplest case, they can be thought of as program invariants and assertions [43], 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 function calls, even if they have side effects (which would be hidden from the user by the testing framework). These are sometimes referred to as "parameterized unit tests" [179].

A simple example of an In Vivo test is one that checks the functionality of an implementation of the Set interface (such as a Vector or ArrayList) 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.

Additionally, many security-related defects only reveal themselves under certain conditions, specifically the configuration of the software together with its current application

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

state [49]. For instance, the FTP server wu-ftpd 2.4.2 assigns a particular user ID to the FTP client in certain configurations such that authentication can succeed even though no password entry is available for a user, thus allowing remote attackers to gain privileges.[2] Here, an In Vivo test may be one that attempts to perform actions that the user should not be able to do with the current permissions; if the test succeeds, and is able to perform those actions, then the vulnerability has been detected. Since In Vivo Testing allows for the execution of tests in a variety of configurations and states, these types of defects are more likely to be revealed.

It is important to note that In Vivo tests are not intended to replace unit or integration tests; rather, we introduce a new testing technique whereby tests are run within the context of an executing application, which may be in a previously untested or unanticipated state. As In Vivo tests are individual functions run inside the application, our approach is like unit testing in the sense of calling individual functions with specified parameters, but it is also like integration testing in that we use the integrated code of the whole application rather than stubs and drivers.

5.1.3 Categories and Motivating Examples

To examine the feasibility of our testing approach, we investigated the documented defects (mostly caught by end-users after deployment) of some open-source applications to see which of them could have been discovered using In Vivo Testing. We considered OSCache [146], a multi-level caching solution designed for use with JSP pages and Servlet-generated web content, as well as another caching solution, Apache Java Caching System (JCS) [5], and Apache Tomcat [7], a Java Servlet container.

We identified five different categories of defects that In Vivo Testing could potentially detect. The categories are listed in Table 5.1. There may be other types of defects that could be found with In Vivo Testing, but these are the ones identified so far.

[2] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1668

1	Unit tests make incorrect assumptions about the state of objects in the application
2	Possible field configurations not tested in the lab
3	Deployment environments not simulated in the lab
4	A user action puts the system in an unexpected state
5	Those that only appear intermittently

Table 5.1: Categories of defects that can be detected with In Vivo Testing

The first category of defects likely to be found by In Vivo Testing are **those in which the corresponding unit test assumes a clean slate**, but the code does not work correctly otherwise. By clean slate, we mean a state in which all objects or structures have been created anew and are modified only by the unit test or functions the test calls, such that the unit test has complete control of the system. Generally unit tests are written in such a way that the entities being tested are created and modified to obtain a desirable state prior to testing [90]. In these cases, the code may pass unit tests coincidentally, but not work properly once executed in the field, revealing defects in both the test code and the code itself. State-based testing [181] or static analysis [62] could be used to look for defects in this category, though these may not be as useful as In Vivo Testing when the system state depends heavily on external systems or user input sequences.

One of the documented OSCache defects notes 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] In this case, the corresponding In Vivo test for testing cache removal may simply add something to the cache, remove it, and then check that it is no longer there. This sequence of operations should work consistently regardless of the state of the cache. A unit test that assumes an empty or new cache would pass; however, when the cache is full, the In Vivo test would fail, revealing a defect that may not have been caught until it affected a user.

Another example of this type of defect can be found in Apache JCS. Here, the method

[3] http://jira.opensymphony.com/browse/CACHE-236

that returns the number of elements in the cache is off by two when the cache is at full capacity.[4] A unit test that simply creates a new cache, adds some number of elements, and checks the size may pass in the development environment if the number of added elements is smaller than the capacity. But an In Vivo test that is executed in the field would detect this defect when it tries to add those elements and thus meets the cache's capacity.

One could argue that, in both cases, a sufficient set of test cases would consider the particular equivalence class (namely, when the cache is at or near full capacity), but it may be easier for the tester to simply specify that these properties should always hold, rather than explicitly creating the infrastructure needed for the test cases that exercise the code in particular states. Although the JCS defect could conceivably be caught with a program invariant (specifically, that the size of the cache after the element is added should be one greater than before it was added), the OSCache defect could not be, since program invariants necessarily do not have side effects, and removing an element would obviously alter the cache's state; however, an In Vivo test could detect this, because side effects are hidden from the user.

The second category targeted by our approach includes **those defects that come about from field configurations that were not tested in the development environment**. For instance, the version of Apache Tomcat that we considered has over 60 different parameters that can be configured, many of which allow for free-text input or unbounded integer values, meaning the entire potential configuration space is huge. We note that a testing approach using a system like Skoll [94, 121] to run tests at the production site before deployment of the software could potentially find some defects in this category, but others will only reveal themselves once the application has been running for a while, and would not be detected prior to the application's deployment and widespread use.

OSCache has around 20 configurable parameters, and one of its known defects falls into this category, too. In this defect, setting the cache capacity programmatically does not

[4] http://issues.apache.org/jira/browse/JCS-16

override the initial capacity specified in a properties file when the value set programmatically is smaller.[5] 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 defect.

The third category of defects concerns **those that come about from deployment environments that were not simulated in the lab prior to release**. Java applications may require testing on multiple platforms with multiple JDK versions and multiple revisions of the application code, possibly with multiple third-party libraries or application servers; this is not always feasible for testing in a single test lab. Additionally, a new JDK, OS, or library may be released after the software is deployed, making testing prior to deployment impossible. For instance, in OSCache certain functionality works fine with Solaris 8 but not Solaris 9, which was released after the version of OSCache in question.[6] By extending testing into the various deployment environments, In Vivo Testing would detect such defects.

The fourth type of defects targeted by In Vivo Testing are **ones that stem from a user action that puts the system in an unexpected state** that would not have been tested. These actions may be legal ones that were simply unanticipated, or illegal actions, e.g., a security violation. This could also happen when data elements in the same process are shared between users, and one user's activities modify some data such that it causes a problem for other users.

For example, in OSCache, an uncaught NullPointerException would appear only after a particular sequence of operations that involves attempting to flush cache groups that do not exist.[7] In this case, an In Vivo test that checks the operation of the flush method would

[5] http://jira.opensymphony.com/browse/CACHE-158
[6] http://jira.opensymphony.com/browse/CACHE-193
[7] http://jira.opensymphony.com/browse/CACHE-173

detect this invalid state because the test would fail, even though that test would succeed in normal "expected" states.

The fifth and final type of defect is **one that only appears occasionally**. These defects may be discovered by simply conducting more testing during the development phase, but the fact that our approach continuously tests the application even after deployment increases the chance of finding such a defect.

Concurrency bugs are a very common type of defect in this category. We noticed one of the concurrency bugs in Apache Tomcat, in which a particular method used in the creation of a session is not threadsafe. If the thread that invalidates expired sessions happens to execute at the same time as a session is being created, it is possible that an uncaught exception would occur because one of the objects being used in the session creation could be set to null by the invalidator.[8] A unit test that is simply testing the creation of sessions is not likely to detect this defect because at that time there may not be any other sessions to invalidate (this is also a case of the first type of defect targeted by In Vivo Testing, in which the unit test assumes a blank slate). However, in the deployment environment, this unit test may fail if the session invalidation thread is cleaning up other sessions at the same time.

We found at least ten such examples in the listing of known OSCache defects. For instance, in one of them, 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.[9] 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.

Note that, in all these cases, In Vivo Testing helps find defects in poorly designed unit tests as much as it does in the applications themselves. Software testers may not anticipate these types of defects when they write their tests, but we hope that by using In Vivo Testing, they will consider a different approach that allows them to test functionality

[8] http://issues.apache.org/bugzilla/show_bug.cgi?id=42803
[9] http://jira.opensymphony.com/browse/CACHE-175

of the application, regardless of its state or environment.

Also, it is conceivable that the defects documented here could have been discovered prior to release of the application given more time, better unit tests, and a little luck. But these examples demonstrate that a testing methodology that continues to execute tests on an application in the field greatly improves the chances of the errors being detected before affecting an end-user. More importantly, certain defects will in practice only manifest themselves in the field (because of limited time and resources in the testing lab, or because they are heavily dependent on the state), and these are the ones for which In Vivo Testing is most useful.

As mentioned, other testing approaches such as the use of program invariants or data structure integrity checking [52] may not be suitable since those do not allow for side effects, and thus are limited to "read-only" types of tests, but cannot run arbitrary code at arbitrary execution points to check its consequences. And while In Vivo Testing does not seek to replace conventional unit and integration testing, others have noted that in some cases it may be easier for the tester to specify the properties that a function is expected to have, regardless of system state, as opposed to specifically constructing states in which to test based on equivalence classes [179]. In that sense, In Vivo Testing acts as a safeguard against states that were not anticipated during traditional unit testing.

5.1.4 In Vivo Testing Fundamentals

To apply the In Vivo Testing approach, the application vendor must first perform some preparation steps (described in Section 5.3.1), including the instrumentation of the portions of the application that are to be tested in the production environment. After these preparation steps have been performed and the application has been configured to take advantage of In Vivo Testing, it is deployed in its usual fashion: the application user does nothing special and would not even know that In Vivo Testing is being performed. In Vivo testing then works as follows: when an instrumented part of the application is to be executed, with some

probability a corresponding test is then executed in a separate "sandbox" that allows the test to run without altering the state of the original application process. The application then continues its normal operation as the test runs to completion in a separate process, and the results of the test are logged. Note that the tests are only invoked as a result of the execution of the code they are testing, so that commonly used code is tested more often.

Although the In Vivo Testing approach is a general testing approach suitable to most types of applications, it is most appropriate for those that produce calculations or results that may not be obviously wrong, and do not otherwise make the error obvious, such as crashing or hanging. For instance, in most of the caching examples above, the user would not notice that the cache is acting incorrectly, as the data would still be usable and may appear to be correct; however, in those examples the caches are not working correctly and/or as efficiently as they should. Applications that include complex calculations may also benefit from In Vivo Testing because the user may not know whether the calculations are obviously incorrect, but defects in the implementation could cause slightly erroneous results (the tests used in Metamorphic Runtime Checking, presented in Chapter 4, are thus an example of In Vivo tests). Systems that have complex states that perhaps could not be anticipated in advance are other good candidates for In Vivo Testing, which is designed to execute tests in such situations.

5.2 Model

Figure 5.1 shows a representation of the model for In Vivo Testing. When a function is about to be executed, the framework determines whether or not to run a test, based on a number of factors such as the current number of tests being conducted, the system load, etc. If a test is to be run, a new environment is created as a copy of the original, so that the test runs in the same application state as the function it is testing. The function and its corresponding test are then run in parallel, i.e., the test does not preempt the rest of the

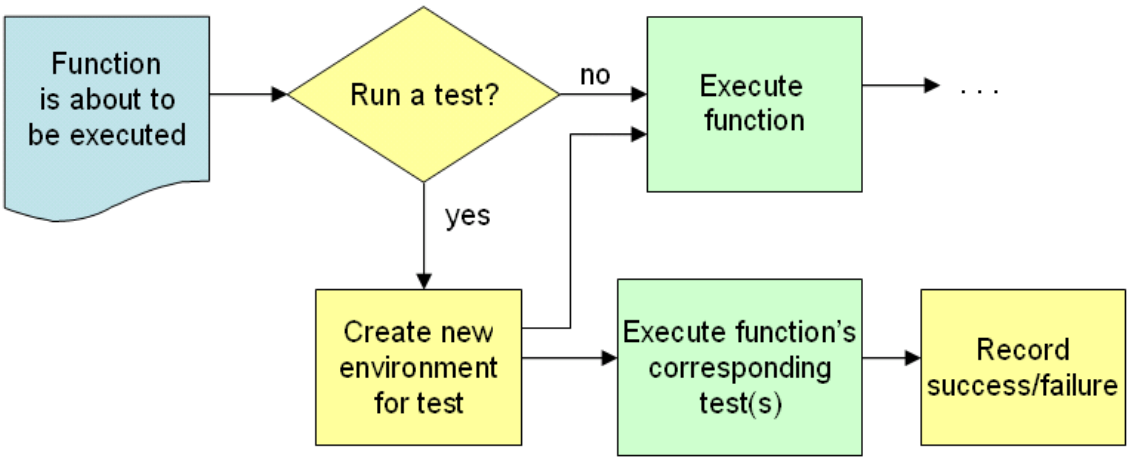

Figure 5.1: Model of In Vivo Testing

application (note, however, that the test could conceivably be executed at a later time and/or on a different machine, as long as the state in which the test runs is the same as that of the function it is testing; this is left as future work) . When the test finishes, its success or failure is recorded. Meanwhile, the original application continues executing as normal.

5.3 Architecture

The prototype In Vivo Testing framework, which we call *Invite* (IN VIvo TEsting framework), has been implemented for Java and C applications and has been designed to reuse existing test code and to allow for the creation of In Vivo tests, while not imposing any restrictions on the design of the software application. This section discusses the specific details of the implementation.

5.3.1 Preparation

Here we describe the steps that a software vendor would need to take to use the Invite framework. It is important to note that these steps do not require any modification or special constraints on the design of the software application itself; the development of

any new test code and the configuration of the framework would be done *a priori* by the vendor who plans to distribute an In Vivo-testable system, and not by the customer in whose environment the tests run.

Step 1. Create test code. If unit and integration tests already exist, it is certainly possible to use the Invite framework without writing a single line of new code. By shipping these tests with the application and then running them In Vivo as the application executes in the field, it is clear that defects that appear infrequently are much more likely to be revealed purely by increasing the number of times the tests are executed. Furthermore, it is also clear that this approach will help find defects that only appear in certain configurations or environments, since the tests will run in a broad variety of settings [121]. Thus, one can take advantage of In Vivo Testing even without writing any new code.

To get the most out of In Vivo Testing, however, application developers should create In Vivo tests, introduced above in Section 5.1. These tests are designed to check properties of the application that should hold true regardless of its state, and these are most likely to reveal defects that were not found (or could not have been found) in the development environment.

To create In Vivo tests, the software vendor must ensure that the test functions reside in the same class (for Java) or file (for C) as the code they are testing (or in a Java superclass). Also, the In Vivo test for a function `foo` should be a function called `__INVtest_foo`, which indicates whether or not the test passed, so that Invite can log the result and possibly take some appropriate action. The parameters to `__INVtest_foo` should be the same as those to the original function `foo`, so that the actual arguments can be used when testing. Additionally, rather than create new objects or data structures to test in the In Vivo test functions, those functions should use existing ones, since the goal of In Vivo Testing is that, when the test is run in the field, it is using the data that has been modified over the course of the application's execution.

Figure 5.2 shows a simple In Vivo test that could be used in a Set implementation in Java.

Upon invocation of the `add` method with an Object parameter, for instance, `__INVtest_add` is called and the argument is passed to it as `testObj`. Because this test method resides in the same class that defines the `add`, `remove`, and `contains` methods, it uses the object reference `this` to call functions on itself.

```
public boolean __INVtest_add(Object testObj) {
   this.add(testObj);
   if (this.contains(testObj) == false) return false;
   this.remove(testObj);
   return (this.contains(testObj) == false);
}
```

Figure 5.2: Example of In Vivo test

It may be possible to automatically detect properties that could be used in the creation of In Vivo tests, and of course developers may have their own notion of what properties of the function should hold, regardless of the application state. These properties may relate not only to functional correctness, but also to security, performance, etc.

For instance, Figure 5.3 shows an In Vivo test that checks for "security invariants" [21], i.e., security-related aspects of the program that are always expected to hold. The function being tested is the `login` function, which takes a username and password as arguments. On line 2, the In Vivo test attempts to log in as the user. If the login attempt is unsuccessful (line 3), then the test attempts to read some data (line 4). Here, the security invariant is that "a user who is not logged in should not be allowed to read data". In the test, if the user is not logged in and the reading of data is successful (line 5), then the In Vivo test returns 0 to indicate that the test failed, i.e., that the security invariant was violated. On lines 7-8, the username and password are set to null, and another login attempt is made on line 9. The security invariant in this case is "a user should not be allowed to log in with empty username and password". If the login attempt succeeds (line 10), then the test returns 0, indicating that the security invariant was violated and that the test failed. Otherwise, it returns 1 (line 11) to indicate that the test passed.

Another approach to creating In Vivo tests is to build upon already-created unit tests,

```
 1 int __INVtest_login(char* username, char* password) {
 2    int loginSuccessful = login(username, password);
 3    if (loginSuccessful == 0) {
 4       int readSuccessful = readData();
 5       if (readSuccessful) return 0;
 6    }
 7    username = NULL;
 8    password = NULL;
 9    loginSuccessful = login(username, password);
10    if (loginSuccessful) return 0;
11    else return 1;
12 }
```

Figure 5.3: Example of In Vivo test

modified so that they use existing objects and data structures, rather than constructing new ones. Figure 5.4 shows such a unit test in the JUnit [89] style for Java applications. It is clear that there are only small changes required to convert this into the In Vivo test in Figure 5.2: (1) the test method has been moved into the same class as the one it is testing; (2) the name of the test method has been changed to match that of the method it is testing; (3) the parameter to the test method matches that of the original method, and the parameter is used in the testing; (4) the return type of the test method has been changed, and a return statement is used instead of an assert; and (5) the reference to the object being tested (in this case, the Set) in the test function is now "this" instead of a newly-created object. Future work could look into the potential of automating this conversion.

```
private Set set;
@Before public void setUp() {
   set = new MySetImpl();
}
@Test public void testAddRemoveContains() {
   Object testObj = new Object();
   set.add(testObj);
   assert(set.contains(testObj) == true);
   set.remove(testObj);
   assert(set.contains(testObj) == false);
}
```

Figure 5.4: Example of JUnit test

As noted above, the modification of unit tests into the style of In Vivo tests is *not* strictly a requirement for using the Invite framework. Existing unit and integration tests can be used without any modifications whatsoever, and the different types of tests are not mutually exclusive. However, our intention is to demonstrate that it is possible to create In Vivo tests only with small changes to existing unit tests.

Step 2. Instrument classes. In the next step, the vendor must then select the functions in the application under test for instrumentation. Aside from acting as jumping off points for the tests, the instrumented functions are also the same ones that will be tested by the Invite 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 is specified in a configuration file. To achieve instrumentation, a pre-processor "wraps" the original code with the calls to the framework and to the test code; this is further described below in Section 5.3.2. Although this requires recompilation, this restriction could be lifted by use of a system like Kheiron [69], which would dynamically insert the test harness code into the application after it is already compiled.

Step 3. Configure framework. Before deployment, the vendor would then configure Invite with values representing, for each instrumented function, the probability ρ with which that function's test(s) will be run. This configuration specifies the name of the function and the percent of calls to that function that should result in execution of the corresponding tests (if a function is associated with multiple tests, these are all specified separately). The file is read at the application's startup-time (not at compile-time) so it can be modified by a system administrator at the customer organization if necessary. A "DEFAULT" 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 all In Vivo Testing for all but the specified functions. To disable testing for all functions in the application, the administrator can simply put "DISABLE" in the first line of the file.

Note that if function `foo` is called twice as frequently as function `bar`, and both have equal ρ values, then `__INVtest_foo` is going to be called twice as frequently as `__INVtest_bar`, which we feel is desirable since that function should be tested more often since it is called more often. However, it may also be the case that more defects may reside in less frequently-executed (and presumably less frequently-tested) functions, so the developer has the option of setting different values of ρ for the individual functions; this is discussed further in Future Work (Section 7.2).

Step 4. Deploy application. It is assumed that the application vendor would ship the compiled code including the tests and the configured testing framework as part of the software distribution. However, the customer organization using the software would not need to do anything special at all, and ideally would not even notice that the In Vivo tests were running.

5.3.2 Test Execution

Figure 5.5 shows the pseudocode for an instrumented function `foo`. When the function is called (line 8), a check is performed (line 9) to see whether an In Vivo test should be run at this point, using the specified percentage value ρ for that function, as well as other parameters described below. The purpose of running a function's corresponding test function is so that the test is executed at the same point in the program (the same state) as the function itself, 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 it is determined that a test should be run, Invite then forks a new process (line 10), 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 In Vivo test will not affect the "real" application, since the test is being executed in a separate process with separate memory. Performing a fork creates a copy-on-write version of the original process, so that the process running the test has its own writable memory area and cannot affect the

```
 1  /* original function */
 2  int __foo(int p1, int p2) { ...  }
 3
 4  /* In Vivo test function */
 5  boolean __INVtest_foo(int p1, int p2) { ...  }
 6
 7  /* wrapper function */
 8  int foo(int p1, int p2) {
 9    if (__should_run_INVtest_foo()) {
10      create_sandbox_and_fork();
11      if (is_test_process()) {
12        if (__INVtest_foo(p1, p2) == false) fail();
13        else succeed();
14        destroy_sandbox();
15        exit();
16      }
17    }
18    return __foo(p1, p2);
19  }
```

Figure 5.5: Pseudo-code for wrapper of instrumented function

in-process memory of the original. Once the test is invoked (line 12), the application can continue its normal execution, in which it invokes the original "wrapped" function (line 18), while the test runs in the other process. Note that the application and the In Vivo test run in parallel in two processes; the test does not preempt or block normal operation of the application after the fork is performed.

In the current implementation of Invite, 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 limitation, we have integrated Invite with a "process domain" sandbox [147] to create a virtual execution environment that isolates the process running the test and gives it its own isolated view of the file system and process ID space. However, this virtualization layer does not include external entities (see Future Work in Section 7.2 for more details), and the overhead of creating the sandbox may be prohibitive to running tests in the field in the general case. Thus, this type of sandbox is only used when indicated by a special notation

in the configuration file.

When the test is completed, Invite logs whether or not it passed (Figure 5.5, lines 12-13), and the process in which the test was run is terminated (lines 14-15). Invite provides a tool for analyzing the log file and providing simple statistics like the number of tests run, the number that passed/failed, and a summary of the success/failure of each instrumented function's corresponding test(s). Additionally, results can be reported back to a central server (presumably at the vendor's location), as described for the Amsterdam framework in Section 3.2.6. The results of the tests could then be processed, and configuration parameters (like the frequency of test execution or even the list of functions to test) could conceivably be modified and sent back to the application instance [145]; the implementation of this part of the solution is left as future work.

Unlike other testing approaches that test the application as it is running [54, 126], Invite avoids the "Heisenberg problem" of having the test alter the state of the application it is testing. This is one of the major contributions and differentiating characteristics of the In Vivo Testing approach.

5.3.3 Scheduling Execution of Tests

We have also considered other policies for determining how frequently unit tests should be run, aside from the static configuration value. The relative effects of these different policies, however, are outside the scope of this thesis. 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 function 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 functions where more defects were found during lab testing and/or more field defects were reported, so as to increase the likelihood of finding a defect in a potentially flawed function.

5.3.4 Configuration Guidelines

In order to help a system administrator or vendor understand the configuration's impact on performance and testing, Invite periodically records to a log file the total number of In Vivo tests that have been run, the average time each test takes, and the number of tests run per second. All of these statistics are tracked in total for the entire application, but also for the separate functions, since they may have different ρ values. From this data, it is then possible to estimate how altering the value of ρ will affect the system's performance and number of tests executed.

Specifically, the rate of tests run per second is proportional to ρ: for instance, to double the frequency of execution of a particular test, simply double the function's ρ value. This simple calculation will help guide how to adjust ρ so as to execute more (or fewer) tests for a given function, assuming constant usage of that function over time.

To estimate the performance overhead caused by the unit tests, one can multiply the number of In Vivo tests by the average time each takes to see what additional time is being spent running those tests. Then, by calculating the effect that ρ has on the number of tests being run per unit time, one can then calculate the additional overall time cost of increasing or decreasing ρ. We surmise that, in practice, the ρ values would presumably be very small. However, these are heavily dependent on the number of instrumented functions, the frequency with which they are called, the desired amount of testing to be performed, and the acceptable performance degradation. We discuss more performance issues in Section 5.5.

Even with small values of ρ, there is a danger of the test processes flooding the CPU, so that the original process (as seen by the user) is prevented from running. To address this, the Invite framework can be configured to limit the maximum number of concurrent tests that the system is allowed to execute at any given time. This prevents the testing framework from launching so many simultaneous tests that they flood the CPU and essentially block the main application. A system administrator can also set a maximum allowable performance

overhead, so that tests will be run only if the overhead of doing so does not exceed the threshold. The system tracks how much time it has spent running tests compared to how much time it has been running application code, and only allows for the execution of tests when the overhead is below the threshold.

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 currently 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. For instance, on a quad-core machine, one core could be executing the application, allowing for up to three simultaneous tests, each on a separate core, so that none of them would preempt the original application process, or each other.

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

5.3.5 Distributed In Vivo Testing

Although the Invite framework can be configured so that the performance impact is not great enough to be noticeable, reducing the overhead comes at a cost of reducing the number of tests. Given that many applications are used in numerous installations, it follows that a group of applications instrumented with Invite would be able to run many tests globally, without putting too much testing load on a single instance. This section introduces the idea of *distributed In Vivo Testing*: applying the In Vivo Testing approach to an "application community" (a collection of autonomous application instances running across a wide-area

network) [108], where the size of the community can be leveraged to detect defects and reduce overhead.

Applying a distributed approach to In Vivo Testing is motivated by two reasons. First, amortizing the workload over many instances tackles the issue of performance overhead, without sacrificing the quantity of tests being conducted globally. Second, In Vivo Testing relies upon testing as many permutations of state as possible, in the hopes that one will be encountered that causes an error to be revealed in the code. Having a community of applications collaboratively working together increases the possibility that at least one instance will find such a fault-revealing state.

To distribute the In Vivo testing load, we extended the original Invite framework with a distributed component that follows a simple server-client model, where each application under test includes the Invite client. The Invite server is a separate standalone component that runs on a separate machine, most likely hosted by the vendor.

The distributed Invite framework seeks to reduce the overhead of each Invite client by reducing the number of tests each instance has to run while maintaining the same global quantity. This distributed effort is coordinated by the central Invite server. The basic protocol is:

1. The Invite server is initialized and ready to receive Invite client requests.

2. When an instance of the instrumented application begins, it logs into the Invite server for the first time and registers itself as an Invite client as part of its initialization process. The Invite client receives a unique client id, a list of tests to run, a probability ρ to run the tests, and a time t to reconnect to the Invite server.

3. The application under test executes normally, except that with probability ρ the Invite client (randomly) executes one of the assigned tests using the In Vivo Testing approach.

4. At time t the Invite client connects to the Invite server with its identification, sends

the results of its testing, and receives a new time *t'* to reconnect, as well as possibly a new list of tests and/or a new probability ρ'. In this way the Invite server can dynamically adjust to changes in community size by modifying the values it assigns.

The Invite server handles all the bookkeeping of maintaining client ids, distribution and assignment of the test suite to the community, the results of each test and the Invite client that executed it, and the reconnect times. The full test suite is intelligently partitioned by the Invite server based on the size of the community, and these partitions are rotated periodically (as in [121] and [145]). There is also a simple console client that allows a developer to query the Invite server for reports. Currently, the server can report the number of clients in its community, the number of tests run by each client, and the results of each individual test.

5.4 Case Studies

Given the motivating examples listed in Section 5.1, we sought to apply Invite to some of those applications to demonstrate that In Vivo Testing would have quickly detected those defects, even assuming the presence of sufficient unit tests that could be used in the development environment.

We first investigated OSCache 2.1.1, which contained three of the known defects listed in Section 5.1. Unfortunately the unit tests that are distributed with that version of OSCache do not cover the functions in which those defects are found, so we asked a student (who was not aware of the goals of this work) to create 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 the student had created the tests assuming a clean state that he could control. This took a total of two hours.

We then asked the same student to develop In Vivo tests, using those unit tests as a starting point; it took less than one hour to complete this task. Next we instrumented the

corresponding classes in OSCache with the Invite framework. 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.

A similar experiment was conducted with Apache JCS version 1.3. Here we were looking for a defect that only appeared when the cache was at full capacity, and this defect was revealed in less than one hour, but again may have appeared sooner in the real world.

Although these defects were discovered in our own testing environment (as opposed to a deployment environment), these examples demonstrate that certain intermittent defects or those that only are revealed under certain circumstances may not be revealed in traditional unit testing, but would be detected with In Vivo Testing. More importantly, these case studies demonstrates the technical feasibility of our approach and is indicative of its efficacy in such situations.

5.5 Performance Evaluation

To understand the performance impact of the In Vivo Testing approach, we revisited the machine learning applications used in the studies in the previous chapters, and measured the impact of the Invite framework on the time it took to conduct testing. For the applications C4.5, MartiRank, SVM, and PAYL, we instrumented the functions listed in Table 5.2 with the Invite framework, and varied the probability ρ with which an In Vivo test would be executed while the application ran. In order to get a better measurement of the upper bounds of the effect of In Vivo Testing, we did not place any limits on the maximum

allowable performance overhead or on the number of simultaneous test processes, though the framework still tracked these values.

Application	Function	Function Description
C4.5	FormTree	Creates decision tree
C4.5	Classify	Classifies example
MartiRank	pauc	Computes "quality" [76] of a ranking
MartiRank	sort_examples	Sorts set of examples based on given comparison function
MartiRank	insert_score	Inserts a value into an array used to hold top N scores
PAYL	computeTCPLenProb	Computes probability of different lengths of TCP packets
PAYL	testTCPModel	Returns distance between an instance and corresponding "normal" instance in the model
SVM	distributionForInstance	Estimates class probabilities for given instance
SVM	buildClassifier	Creates model from set of instances (training data)
SVM	SVMOutput	Computes output (distance from hyperplane) for given instance

Table 5.2: Instrumented functions for In Vivo performance test

Tests were conducted on a server with a quad-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. For C4.5, we used the "ad" dataset from the UCI machine learning repository [138]; for MartiRank, we used one of the data sets from the experiments in Section 4.5.1; for SVM, we used the "iris" data set [138]; and for PAYL, we used data collected on our department's LAN over a one-hour period.

Table 5.3 and Figure 5.6 show the results of the experiment, with the baseline measurement (no instrumentation), ρ equal to 0% (with the functions instrumented but no tests executed), 25%, 50%, 75%, and 100% (with all instrumented function calls resulting in tests).

The linear nature of the resulting graphs indicates that, as one would expect, the overhead increases linearly with the number of tests that are executed. The slope of the

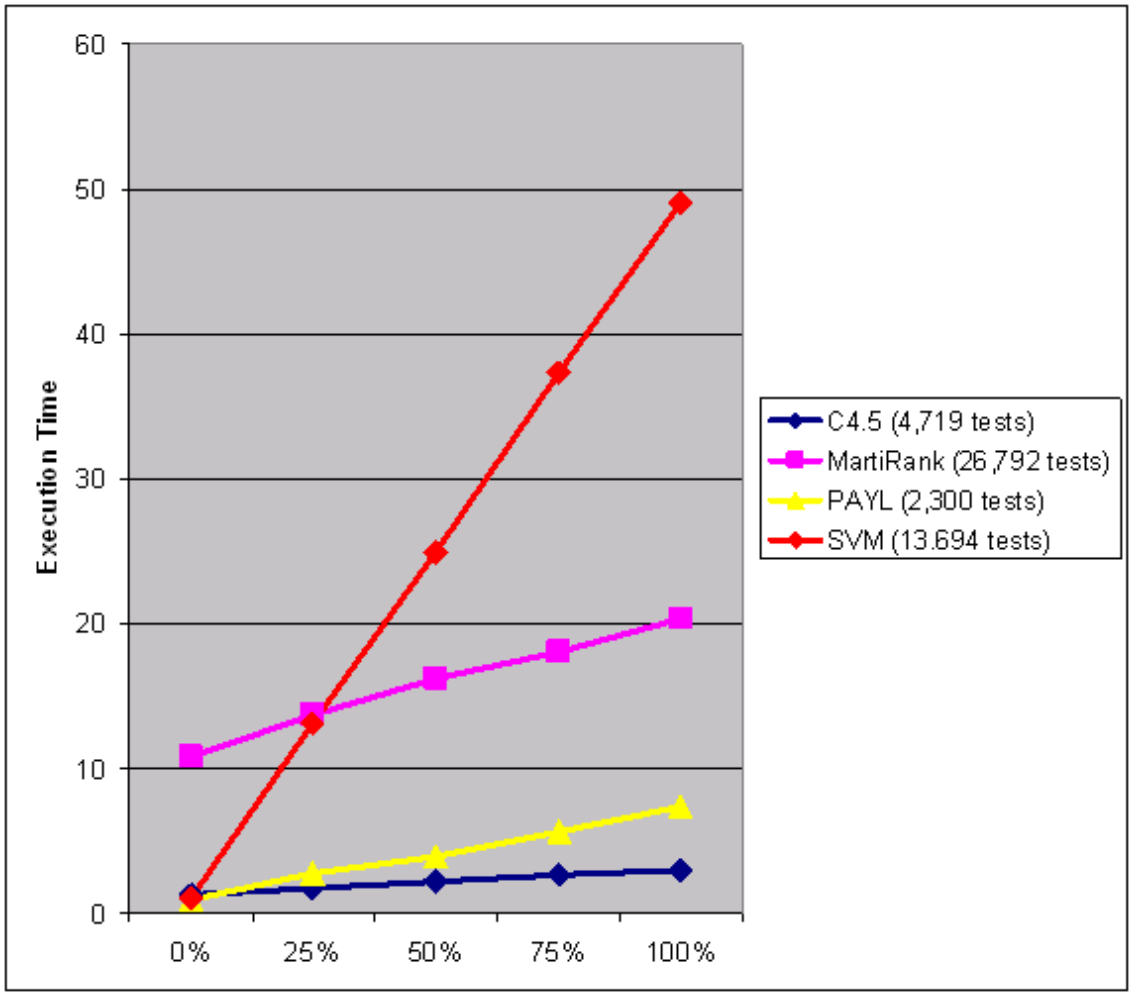

Figure 5.6: Performance overhead caused by different values of ρ for the different applications.

Application	# tests	Baseline	ρ = 0%	ρ = 25%	ρ = 50%	ρ = 75%	ρ = 100%
C4.5	4,719	1.3s	1.3s	1.7s	2.2s	2.6s	3.0s
MartiRank	26,792	10.6s	10.8s	13.7s	16.2s	18.1s	20.4s
PAYL	2,300	1.1s	1.1s	2.8s	3.9s	5.6s	7.3s
SVM	13,694	0.9s	0.9s	13.1s	24.9s	37.4s	49.1s

Table 5.3: Results of Performance Tests. The five rightmost columns indicate the time to complete execution with different values of ρ.

lines results from a combination of the number of tests that are run and the implementation language: the line for SVM is very steep because many tests were run and the overhead is greater for Java applications (since Java does not have any "fork" utility, it needed to be implemented via a Java Native Interface call, which added extra overhead); the line for C4.5 is less steep because fewer tests were run and there is less overhead for C.

On average, the performance overhead for the Java applications was around 3.5ms per test; for C, it was only 0.4ms per test. These numbers are consistent with the overhead of the Columbus framework for Metamorphic Runtime Checking (Section 4.6) and indicate that additional instrumentation (e.g., checking for a maximum number of simultaneous test processes) is negligible.

Despite the large overhead incurred by running numerous tests very frequently, the results show that incurring an overhead of just one second still achieves thousands of tests for a single application instance written in C, and hundreds of tests for a Java application. This experiment demonstrates that it is possible to gain the benefits of In Vivo Testing with limited performance overhead.

5.6 More Efficient Test Execution

One of the issues of the Invite implementation described in Section 5.3 is that there is a possibility that a function's In Vivo tests will run multiple times in the same application state, potentially causing unnecessary overhead. In this regard the following question arises: "Can the technique be made more efficient by only running tests in application states it has

not seen before?" Invite currently runs tests probabilistically or according to the testing load; however, certain states may unnecessarily be tested multiple times. If the results of the test are deterministic, in that they depend *only* on the system state, then we can possibly make Invite more efficient by having it run tests only in previously-unseen states.

Of course, these approaches are not mutually exclusive. For example, we could run tests only in previously-unseen states *and* with some probability, so that when we encounter a state that has not been tested, there is only a specified probability that we will actually run the test. The downside of that, however, is that some states may never be tested, depending on the probability and the number of times they are encountered. Thus, in this section we investigate whether we can improve the efficiency of In Vivo Testing by reducing any unnecessary overhead, but without making any sacrifices regarding the set of states in which tests are run.

5.6.1 Analysis

An approach designed to increase the efficiency of In Vivo Testing (or any approach, for that matter) based on running tests only in previously-unseen states is heavily dependent on the assumption that a given function will, in fact, run in the same state multiple times. In the best case, if the function *always* runs in the same program state, then the In Vivo test will only be run once, i.e., the very first time, and the performance overhead will approach the theoretical minimum of never running any tests, give or take a little bit of overhead from the instrumentation. In the worst case, if the function *never* runs in the same state, then In Vivo tests will run for every invocation of the function, which will incur worse performance overhead than the "standard" In Vivo approach, since not only are test functions being run, but there is extra overhead from determining whether the state had been seen before. It follows, then, that there must be some percentage of previously-seen states such that the new approach will, in fact, be more efficient.

The theoretical analysis is rather straightforward. We define the *Distinct State Percent-*

age (*DSP*) as the number of distinct states in which a function is called, divided by the total number of times the function is called. We define the *Repeat State Percentage* (*RSP*) as 1 - *DSP*. For example, if a function is called in states A, B, A, C, B, C, A, and A, then the *DSP* is 3/8 (since it was called 8 times and had three distinct states) and the *RSP* is 1-*DSP* = 5/8 (since five of the times, the function was called in a state it had already seen).

We also define the following:

- t_s = the time it takes to create the sandbox in which the In Vivo test will be run
- t_d = the time it takes to determine whether the function had already been run in the current state
- t_u = the time it takes to update the data structure storing previously-seen states
- N = the number of times the function is called

Given these definitions, we can simply calculate the overhead from "standard" In Vivo Testing (assuming that the tests are executed on every function invocation) as:

$$T_{invivo} = N*t_s.$$

Note that the overhead T_{invivo} could be reduced by a constant factor if tests are only executed with some probability ρ. However, this would mean that tests may not be executed in some application states. Our evaluation here is based on the assumption that it is desirable to conduct tests in all observed application states during program execution, which, after all, is the point of In Vivo Testing.

We can also calculate the overhead from the suggested new approach, in which tests are only conducted in states that have not already been encountered. The time taken when tests are run in previously-unseen states is:

$T_{unseen} = N*DSP*(t_d + t_u + t_s)$.

The time taken when tests are not run because the state had already been seen is simply:

$T_{seen} = N*RSP*t_d = N*(1 - DSP)*t_d$.

The total overhead for the suggested new approach is their sum, $T_{unseen} + T_{seen}$.

To achieve the benefits of running tests only in states that have not previously been seen, we seek a low *DSP* such that the overhead for running tests only in previously-unseen states is less than that of running tests in every state, i.e.:

$T_{unseen} + T_{seen} \leq T_{invivo}$.

Replacing with the formulas above and solving for *DSP*, we get:

$N*DSP*(t_d + t_u + t_s) + N*(1 - DSP)*t_d \leq N*t_s$

$\implies DSP*(t_d + t_u + t_s) + (1 - DSP)*t_d \leq t_s$

$\implies DSP*(t_d + t_u + t_s) + t_d - DSP*t_d \leq t_s$

$\implies DSP*(t_u + t_s) + t_d \leq t_s$

$\implies DSP*(t_u + t_s) \leq t_s - t_d$

$\implies DSP \leq (t_s - t_d) / (t_u + t_s)$

If we can construct a solution such that the time to do a lookup (t_d) or update (t_u) is much less than the time to create a sandbox (t_s), we can see that the right side of the inequality comes close to 1. This means, then, that even for a *DSP* of almost 100%, i.e., even if almost all of the states in which the function runs are distinct, then it still is better to incur the overhead of checking the state and only run tests in states that have not previously been encountered.

5.6.2 Implementation

In developing a new, more efficient implementation of the Invite testing framework, a number of questions immediately arise:

- How do we define a "state"?
- How do we represent the state?
- How can we quickly determine whether the state has already been seen?
- In practice, is any performance gain from running tests only in previously-unseen states outweighed by the overhead of the instrumentation required to track the states?

The following subsections discuss our new prototype, and the implementation decisions that were made in answering these questions.

Determining Function Dependencies

For our purposes, we define the "state" of the application at any given point during its execution as "the values of all variables that are in scope at that point". We acknowledge that this definition is somewhat limiting in that it does not include the state of the *entire* process heap or stack, or the program counter, but we expect that these would be too complex to represent in a format that can be represented and compared efficiently enough to meet our goals. We also do not include external elements such as the state of other

processes, the underlying virtual machine and/or operating system, the file system, etc., for similar reasons. If any In Vivo test relies on these, then this feature of Invite can simply be disabled for the given test, so that it executes regardless of the system state.

Note that a given function may not rely on *all* variables that are in scope at that point in the program's execution. Thus, in determining whether a function has already been executed in the current program state, we only need to consider the variables on which that function depends, i.e., that are read during the function's execution.

To determine which variables a function uses during its execution, we developed a simple pre-processor to parse the source code. For a given function, the pre-processor returns a list of all the global variables (i.e., those declared outside the function definition) that the function uses, and also determines which of the parameters the function depends on, since it may not actually use all of them. Alternative approaches would have been to use data dependence analysis or data flow analysis, but simply parsing the source seemed to be the easiest solution, given that we only need to identify the global variables that are read in the function, and do not need to enumerate all possible values or determine how the variables came to get their respective values at that particular point. Also, although this approach does not detect aliases (i.e., two variables that refer to the same piece of data), we are not concerned with modifications to variables, only with listing the variables that are read during the function's execution.

```
1 int f(int p1, int p2, int p3) {
2   int k = 8;
3   int t = a + 1;
4
5   if (p1 > p2) return k + t;
6   else return p2 - t;
7 }
```

Figure 5.7: Sample function. The Invite pre-processor scans the function looking for variables, to determine the function's dependencies.

Figure 5.7 shows a simple function that can be scanned using the pre-processor. On

line 1, the parameters p1, p2, and p3 are specified; at this point, they are not yet added to the dependency list, since we do not know for certain that the function will actually use them (though it is admittedly rare that they would not be used). On line 2, the local variable k is declared, but because it is assigned a constant value, there is no dependency on this line, either. On line 3, the local variable t is declared; this statement uses the global variable a, which we assume to be declared elsewhere. Because a is on the right side of the assignment, i.e., its value is read, we can add a to the dependency list. In line 5, the conditional compares p1 to p2; thus, because those values are read, they are added to the dependency list. Also on line 5, the return value uses k and t, but we know that these are both local variables, thus there is no extra dependency. Line 6 does not use any new variables so nothing is added this line. Once we reach the end of the function on line 7, we know that the function f depends on the parameters p1 and p2 (line 1), as well as the global variable a (line 3); as it turns out, the function does *not* use the parameter p3.

Given this list of dependencies, we can then claim that, at the point when f is called, only the values of p1, p2, and a will affect its outcome; if those three variables are the same for additional executions, the output of f will not change, nor will the result of the corresponding In Vivo test.

```
 1  int f1(int p1, int p2, int p3) {
 2    int k = 8;
 3    int t = a + 1;
 4
 5    if (p1 > p2) return g(p3);
 6    else return p2 - t;
 7  }
 8
 9  int g(int p) {
10    int m = p * b;
11    return m * m - b;
12  }
```

Figure 5.8: Example of two functions, one of which inherits the set of dependencies from the other.

Now consider the code in Figure 5.8, in which the function `f1` is the same as `f` from the previous example, except that it calls the function `g` on line 5. To determine the dependencies of `f1`, when scanning line 5, we then need to determine the dependencies of `g`. We can see on line 10 that `g` uses the parameter `p` and the global variable `b`; those are the only dependencies of `g`. When that dependency list is returned to `f1`, the parameter `p` is replaced with the argument `p3`. Thus, the overall dependency list for function `f1` becomes: parameters `p1` and `p2`, and global variable `a`, for the reasons described in the previous example; global variable `b`, inherited from function `g`; and parameter `p3`, which was passed as an argument to `g`.

Note that this approach works for other data types besides primitives, including arrays and values referred to by pointers.

In situations in which the pre-processor does not have access to the source code, e.g., if the code makes a system call or uses some external library, then it is impossible to know for certain what the dependencies are, and thus this approach cannot be used. In these cases, the In Vivo tests will be run regardless of the current application state.

Representing States

Once we know the variables on which a function depends, we then need a way of representing the state so that it can be compared to other states to determine whether it has previously been encountered. We can at this point consider the state as a map of a set of variables to their corresponding values. Comparing the sets of values can be time consuming (at least O(n), assuming we know the ordering of the elements to compare) if done element-wise; we require a fast way of comparing the sets, ideally with no false positives (thinking two sets are the same, when actually they are not) or false negatives (thinking two sets are not the same, when actually they are).

In the best case, if we assume that the elements of the sets are numerical, then we can attempt to devise a hashing function such that every set has a distinct value. This would

allow us to effectively represent different program states with a single number.

A hashing function that meets this criteria is a Cantor pairing function [161], which assigns one distinct natural number to a pair of natural numbers. Note that this function has one key characteristic that is crucial in our state formalization, in that it is simple yet effective as the implementation is simply:

`f(k1, k2) = (1/2)(k1 + k2)(k1 + k2 + 1) + k2`.

Using this mathematical tool, we can now take a set of values in the function's dependent state and create a single distinct value, which is critical in determining whether the state had previously been seen. The method for achieving this is to recursively apply the Cantor function to the values, i.e., `f(a, f(b, f(c, ...)))`. This can be done for array elements or by values referred to by pointers, as well.

Tracking Execution States

Given that we have a distinct representation of each execution state, we can then select an efficient data structure to determine whether the function has already been called in the given state, by comparing it to those that it has already seen. We started by investigating the use of a hash table, but a hash table is O(n) in the worst case (where n is the number of elements, i.e., the number of states already seen), and we were hoping for something that would give a better guarantee.

We also considered using a Bloom filter, which is O(1), but a Bloom filter allows for false positives, in that we may think we have already seen a state before, even though we have not. This is not desirable for In Vivo Testing, because it might mean that tests are not executed even though the state has not already been seen. We could, however, allow for false negatives, which would have the result of running tests in previously-seen states; this is not ideal, but at least we do not miss the opportunity to run tests in states that have not previously been encountered.

Our investigation led us to a data structure called a Judy Array[10], which has been proven to demonstrate the properties that we need in a state-management tool. It is space efficient, in that it is a dynamically allocated structure that will not take up space when simply declared for later use. The Judy Array also has the property of consuming memory only when it is populated, yet can grow to take advantage of all available memory if desired. These are especially important features since potentially all functions in the program will need an array to represent which states have previously been seen. A Judy Array is also speed efficient, and is $O(\log_{256} n)$ for lookup operations [47]. Last, it is scalable: this data structure has the potential to use all the available memory on a machine and also claims to be able to hold from zero to billions of elements [47].

Given the selection of a Cantor function for hashing states and a Judy Array for tracking them, we now state the process by which the In Vivo tests of a given function run using the more efficient Invite framework.

1. A pre-processor is used to read the source code and determine which parts of the state (i.e., which variables) the specified function depends on.

2. Another pre-processor creates the necessary instrumentation in the source code so that In Vivo tests become logically attached to the function that they are testing. This generated code makes use of a function that indicates whether a test has already been run in the current state.

3. When the function to be tested is called, the required parts of the current application state are hashed using the Cantor function, which generates a distinct value for that state.

4. The code then checks the function's corresponding Judy Array to determine whether the value already exists in the data structure. If so, then the state has already been encountered, and no test is run. If the value does not exist in the array, though, the

[10] http://judy.sourceforge.net/

state has never previously been encountered, so the value is added to the array, and the Invite framework is instructed to run the test.

5. At this point, In Vivo Testing continues as normal.

Figure 5.9 shows the pseudocode for the instrumentation of a function `f`, which depends on global variable `g` and parameters `p1` and `p2`.

When the function is called (line 21), a check is performed (line 22) to see whether an In Vivo test should be run at this point. The function that performs this check (line 8) uses the Cantor function, recursively if necessary, to generate a distinct value to represent the parts of the state on which the function depends (line 10). If the Judy Array for that function already contains the state (line 12), then there is no need to run the test again (line 13); otherwise, the state is added to the Judy Array (line 15), and the framework is instructed to run the test (line 16).

If it is determined that a test should be run, Invite then forks a new process (line 23), 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 In Vivo test will not affect the "real" application, since the test is being executed in a separate process with separate memory. Once the test is invoked (line 25), the application can continue its normal execution, in which it invokes the original "wrapped" function (line 31), while the test runs in the other process. Note that the application and the In Vivo test run in parallel in two processes; the test does not preempt or block normal operation of the application after the fork is performed. When the test is completed, Invite logs whether or not it passed (lines 25-26), and the process in which the test was run is terminated (lines 27-28).

5.6.3 Evaluation

Although the theoretical analysis provided above shows that the new In Vivo approach will be more efficient even when the function runs in many different distinct sets, we know that

```
 1  /* original function */
 2  int __f(int p1, int p2) { ...  }
 3
 4  /* In Vivo test function */
 5  boolean __INVtest_f(int p1, int p2) { ...  }
 6
 7  /* Determines whether the state has already been seen */
 8  boolean __should_run_INVtest_f(int g, int p1, int p2) {
 9      /* use Cantor function to get distinct value for state */
10      double value = Cantor(g, Cantor(p1, p2));
11      /* determine whether value is already in Judy Array */
12      boolean alreadySeen = JudyArray_f.contains(value);
13      if (alreadySeen) return false;
14      else {
15         JudyArray_f.add(value);
16         return true; // indicates that test should be run
17      }
18  }
19
20  /* wrapper function */
21  int f(int p1, int p2) {
22    if (__should_run_INVtest_f(g, p1, p2)) {
23       create_sandbox_and_fork();
24       if (is_test_process()) {
25          if (__INVtest_f(p1, p2) == false) fail();
26          else succeed();
27          destroy_sandbox();
28          exit();
29       }
30    }
31    return __f(p1, p2);
32  }
```

Figure 5.9: Pseudo-code for wrapper of instrumented function, in which In Vivo tests are only executed in previously-unseen states

in practice the variables used in the calculations may not actually be constant, and we do not know for certain whether the time to fork a new process is significantly higher than the time to do a lookup and update in the Judy Array implementation.

To demonstrate that the new approach is, in fact, more efficient, we conducted a simple experiment in which we measured the time it took to run an application with no In Vivo tests at all (the theoretical minimum time), the time to run with the "standard" Invite

framework that always executes tests regardless of the state, and the time to run with the new Invite framework, using varying percentages of distinct states. In this study, we used the sandboxes created by simple process forking (rather than creating the more heavyweight virtualization layer) to demonstrate that even a small amount of instrumentation overhead can be mitigated by running tests only in previously-unseen states. The goal is to show that, even when the percentage of distinct states is relatively high, the new approach is still more efficient.

The results we present here are for a C implementation of the Sieve of Eratosthenes algorithm, which is given a single number as its parameter and returns a list of all prime numbers less than that number. We chose this program because it only uses one function, but takes a good deal of time to execute, so that we could get meaningful results over many executions. The experiment was conducted on a multi-processor 2.66GHz Linux machine with 2GB RAM.

For inputs, we used data files consisting of 100 random numbers, so that the function would run 100 times. We generated a number of different files with different percentages of distinct values, ranging from 0% distinct (meaning that all values were the same) to 100% distinct (meaning that all values were different).

Figure 5.10 shows the results of the experiment, using the average running time of 10 executions per data set. As expected, the running times for "always" running the In Vivo tests (as in the standard approach) and the time for "never" running tests are more or less constant; they are not exactly constant because of the different values used in the different data sets. More importantly, we see that "sometimes" running In Vivo tests (based only on previously-unseen states) usually outperforms "always" doing it, even with the additional instrumentation, and even when the percentage of distinct states is as high as around 90%.

The results of this experiment demonstrate that our approach does, in fact, make In Vivo Testing more efficient, assuming the percentage of distinct states in which a function is run is less than 90%. Further analysis will be required, however, to determine how true

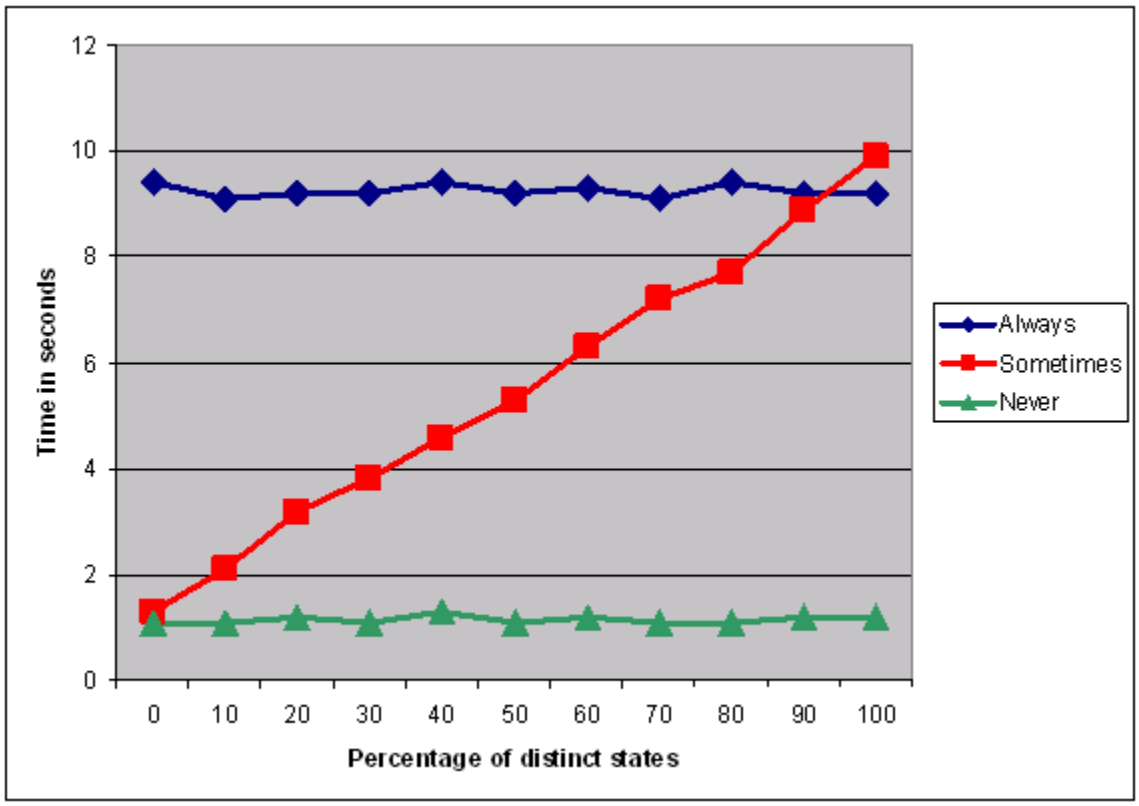

Figure 5.10: Performance impact caused by different variations in the percentage of distinct states.

this assumption is in general.

5.6.4 Limitations

Although it is more efficient to run tests only in states that have not already been encountered, there is a memory cost associated with tracking all the previously-seen states. Regardless of how space efficient the solution may be, a program with many instrumented functions and many distinct program states could have fairly large memory requirements. Future work could assess the practical implications when it comes to additional memory usage.

Aside from the general issue related to memory cost, the specific prototype implementation we have presented here does have some limitations, based on the assumptions stated above. The use of the Cantor function to create a unique hash value for each state does have a practical upper bound in that we cannot store arbitrarily large values in a single variable in C or Java. The Cantor function could conceivably reach the limit of the "double" datatype depending on the values and the number of variables. A solution that scales to arbitrarily large states may not be able to take advantage of the speed of the O(1) hashing function and O($\log_{256}$n) lookup in the Judy Array (thus making its usefulness questionable), or would need to allow for false positives and/or false negatives.

Also, we have made some assumptions regarding the types of variables that can be tracked as part of the state, specifically limiting to primitive datatypes (int, float, char, etc.), arrays of those types of data, and values referred to by pointers, but not arbitrarily complex types such as Objects, structs, and such. This can conceivably be addressed by using type-specific hashing functions, analogous to the Cantor function used for numerical values; however, depending on the uniqueness of the hash codes, this too may introduce false positives, meaning that the system incorrectly believes that the current state has previously been observed. Clearly this would not be desirable, since the result would be that tests or analyses are not performed, even though they should be.

Future work could consider using distributed In Vivo Testing [40] to devise an approach

so that tests are only run in *globally*-unseen states. It may also be possible to distribute the test cases in advance [91, 121], so that a particular instance of the application is not concerned with *all* previously-seen states, only the ones for which it is responsible.

5.6.5 Broader Impact

The ability to quickly determine at runtime whether the current program state has previously been encountered has practical application for many testing and dynamic analysis approaches beyond In Vivo Testing.

For instance, model checking techniques could benefit from knowing whether a function has already been run in a given state. A function may be executed once in a particular state, and then be revisited later via a different execution path, but be set for execution in the same state as before. If it were known that the function had already been checked in that state, then pruning could occur at that point, reducing the number of paths that later need to be investigated. This would be particularly useful for distributed model checking frameworks [91], which require knowing which parts of the state space to distribute, based on which ones have already been considered.

If the set of states in which a function had been executed were known, then checking the set of previously-encountered states could also be used for security testing. We have previously demonstrated [49] that tests in deployed software could be used to check for "security invariants" [21], the violation of which indicate a vulnerability in the software. In that approach, the invariants are test cases written by the tester or developer. However, if the set of acceptable good states (or known bad states) were automatically pre-populated in advance, then it would be easy to determine whether a given function execution should or should not be allowed, given the current state; control could then be sent to a "rescue point" [169] if a known bad state were encountered. Even if the set were not pre-populated, the approach could be used for anomaly detection, i.e., determining whether a particular execution occurs in a state that varies greatly from previous executions.

The application state data collected at runtime could also be sent back to the developers, as it may be useful for the developers of the software to know which functions are being called with what arguments, the number of times the functions are called, the frequency with which they are called in the same state, etc. This information could then be used in regression testing and test case selection [56]. If the sequence of function calls and their corresponding states were also recorded, the data could then be used for fault localization: once an test fails in the field, the developers could investigate the history of function calls in the different application states, culminating with the failed test, and then compare it to executions that did not fail and use techniques such as delta debugging [200] to determine where the defect may have occurred.

Last, such a technique could be used for automatic memoization [142], in that the results (i.e., the output and any side effects) of functions can automatically be cached, thus speeding up the application further. That is, if it can quickly be determined that the function has already been called with the same set of arguments and/or in the same application state, then if the results of the function are already known, there is no need to perform the calculation a second time. Rather, the cached results can be returned, without having to actually execute the function.

5.7 Summary

In this chapter, we have presented *In Vivo Testing*, a novel testing approach that supports the execution of tests in the deployment environment, without affecting that application's state. This technique addresses the problem of not being able to sufficiently test an application in a limited amount of time prior to its release, and can be used to detect defects hidden by assumptions of a clean state in the tests, errors that occur in field configurations not tested before deployment, and problems caused by unexpected user actions that put the system in an unanticipated state

We have also presented *In Vivo tests*, which execute within a running application and test properties of the application that must hold regardless of the state the process is in. Additionally, we have described an implementation of our framework in C and Java, called *Invite*, which adds limited overhead in terms of system performance and code modification.

In doing so, we have proven the third and final part of the hypothesis stated in Section 1.6, that "it is feasible to continue this type of testing in the deployment environment, without affecting the user."

Chapter 6

Related Work

6.1 Addressing the Absence of a Test Oracle

Baresi and Young's 2001 survey paper [15] describes common approaches to testing software without a test oracle. Each of the general approaches from that paper are discussed here. Although other researchers have looked into domain-specific techniques for creating test oracles (e.g., for testing GUIs [120] or web applications [174]), we are not aware of any other significant advances in testing applications without test oracles in the general case, particularly in the domains of interest: machine learning, scientific computing, simulation, and optimization.

6.1.1 Embedded Assertion Languages

Programming languages such as ANNA [111] and Eiffel [124], 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 [43]. In Metamorphic Runtime Checking and In Vivo Testing, the tests can be considered runtime assertions; however, approaches using assertions typically address how variable values relate to each other, but do not describe the relations between sets of inputs and sets of outputs, as we do in Metamorphic Runtime Checking,

or allow for the execution of arbitrary test code, as in In Vivo Testing. Additionally, the assertions in those languages are not allowed to have side effects; in our approaches, the tests are allowed to have side effects (in fact they almost certainly will, since the function is called again), but these side effects are hidden from the user. Last, complex assertions (such as checking for data structure integrity [52]) typically preempt the application by running sequentially with the rest of the program, whereas in Metamorphic Runtime Checking and In Vivo Testing the program is allowed to proceed while the properties are checked in parallel.

Others have reported on the effectiveness of runtime assertions in general [100, 160], and of the invariants created by Daikon in particular [141, 153]. Although some have specifically considered using invariants in place of test oracles [39], the empirical studies presented in Section 3.4, as well as studies independently conducted by Hu et al. [85], demonstrate that metamorphic testing is more effective overall than runtime assertion checking in detecting defects in programs without test oracles.

Note that the techniques described in this thesis do not preclude the use of the embedded assertions. As demonstrated by the experiments in Section 3.4, the combination of metamorphic testing and runtime assertion checking is more effective than either technique alone. Additionally, we suspect in practice that the identification of metamorphic properties (at the system level or function level) would likely occur at the same point in the software development process as the identification of invariants and assertions, making the approaches complementary.

6.1.2 Extrinsic Interface Contracts and Algebraic Specifications

One approach identified in Baresi and Young's paper is the use of extrinsic interface contracts, which are similar to assertions except that they keep the specifications separate from the implementation, rather than embedded within. Examples include languages like ADL [165] or techniques such as using algebraic specifications [45].

Algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (e.g., *pop(push(X, a)) == X* in a Stack implementation) [164], whereas metamorphic properties typically describe how the outputs of a *single* function should relate when the input is changed in a particular way. Additionally, algebraic specifications are typically used to formulate axioms that are then employed to create test cases for particular data structures [66], but are not as powerful for system-level testing in general, since the system itself may not have such properties. Although the tests that are generated do have oracles for the individual cases, the approach cannot be used to create general oracles for arbitrary test cases, i.e., for program operations that do not have associated algebraic specifications, or for the entire system. This is analogous to the fact that Metamorphic Runtime Checking is principally only effective for functions that have metamorphic properties; in our case, though, we also address the use of system-level metamorphic testing to consider metamorphic properties at the application level, which is not a part of the algebraic specification approach.

The runtime checking of algebraic specifications has been explored by Nunes et al. [143] and by Sankar et al. [164], though the implementations of these runtime checking frameworks address the issue of side effects either by cloning objects [143], which raises issues of deep cloning and is only suitable for languages like Java; or by rolling back any side effects [164], which the authors admit may not be possible in all cases. The approaches described in this thesis use a process fork or a virtualization layer, which circumvents these issues by giving the test its own process in which to run. More importantly, these previous works have considered the specification of only small parts of the application, such as data structures, and do not consider properties of functions in general, nor do they investigate mechanisms for using the properties to conduct system testing, as we do here.

Others have looked at the automatic detection of algebraic specifications [80]; the automatic detection of metamorphic properties is outside the scope of this work, but is discussed in Future Work in Section 7.2.

6.1.3 Pure Specification Languages

Coppit and Haddox-Schatz [48] have demonstrated that formal specification languages can be effective in acting as test oracles, by converting the specifications into assertions and invariants that can be checked at runtime. As shown by the empirical studies in Section 3.4, however, metamorphic testing is typically more effective than such an approach, particularly for non-deterministic applications. For instance, consider the case in which a function should return a random number in the range [a, b], but because of a defect actually returns a number in [a, b-1]. An invariant created from the formal specification would not detect this defect because no single execution violates the property that the result should be in [a, b]. However, statistical metamorphic testing would detect this defect because, over many executions, the observed mean and variance differ from what is expected.

Richardson et al. [159] presented a technique for developing a test oracle (as state-based test cases) from a specification, and others have looked into the generation of test oracles from program documentation [41, 150], on the assumption that it represents a reasonable approximation of the specification of the system.

As mentioned earlier, though, the specifications used in these approaches need to be complete in order to be useful [164]. Others have pointed out that specification languages often must make a trade-off between expressiveness and implementability, such that if the language is sufficiently expressive, it becomes hard to write an implementation [182], or to automatically determine whether the software under test is actually adhering to that specification [15]. Further investigation into this phenomenon could determine which, if any, specification languages are most effective from a practical point of view at revealing defects in programs without test oracles.

Approaches to using specification languages as oracles are limited in practice to smaller applications (or small parts of applications), for which it may be easier to write a formal specification. The metamorphic testing approaches presented in this paper should scale to arbitrarily-large applications, assuming metamorphic properties can be identified. Given

that most applications in the domains of interest (machine learning, simulation, scientific computing, etc.) are likely to be developed by scientists and not by software engineers [26], the likelihood that a complete formal specification exists may be quite small. However, metamorphic testing only requires knowledge of what the application is meant to do, not necessarily how it is (or should be) implemented. Thus, the scientists or researchers who develop the application would be in the best position to identify the metamorphic properties and test the software.

6.1.4 Trace Checking and Log File Analysis

Last, it may be possible to perform trace or log file analysis to determine whether or not the program is functioning correctly, if for instance it is conforming to certain properties (like a sequence of execution calls or a change in variable values) that are believed to be related to correct behavior; or, conversely, to see if it is *not* conforming to these properties. Some researchers have investigated the effectiveness of these techniques for specific domains, such as network communication protocols [64] or graphical user interfaces [122], or by using anomaly detection based on models that are assumed to already exist [187]. However, others have demonstrated that the content of the log/trace files must be carefully planned out, often by someone with great knowledge of the application, in order to be of use in the general case [4], especially if it is necessary to learn a new specification or analysis language [196]. That is, the creation of an oracle to tell if the trace is correct can be just as difficult as creating an oracle to tell if the output is correct in the first place, assuming it is even possible at all.

If the correct model of execution cannot be known in advance, a related approach is to observe numerous program executions that are assumed to be correct for a set of given inputs, and then look for deviations or anomalies in subsequent executions with a different set of inputs. Such techniques are common in the security domain, e.g., intrusion detection based on anomalous sequences or patterns of system calls [59, 189], or virus detection

based on anomalous access to the Windows registry [8]. However, these approaches are likely not applicable to the problem of detecting the types of calculation defects investigated in this thesis. Anomaly detection has been used for fault localization [11, 101, 106] or for fault detection using dynamic [75] or static [57] analysis, though these testing techniques typically represent the model of observed (and assumedly correct) execution as a set of program invariants, which in Section 3.4 were shown to be less effective than metamorphic testing in detecting defects in applications without test oracles.

6.2 Metamorphic Testing

The idea of applying metamorphic testing to situations in which there is no test oracle was first suggested by Chen et al. [37], though these works only presented the idea in principle, considering situations in which there cannot be an oracle for a particular application [36, 38], or in which the oracle is simply absent or difficult to implement [28]. Others have applied metamorphic testing to specific domains such as bioinformatics [33], network simulation [35], machine learning [195], and graphics [71], using domain-specific or application-specific metamorphic properties. In our work, we have attempted to identify *general* classes of metamorphic properties that can be used in many different domains, including simulation and optimization (two areas to which metamorphic testing had not previously been applied). Additionally, previous work has conducted metamorphic testing using system-level properties and/or function-level properties that are tested in isolation (akin to standard unit testing), whereas we have introduced a new concept in Metamorphic Runtime Checking, and demonstrated its effectiveness compared to system-level metamorphic testing in Section 4.5.

The study presented in Section 3.4 is most similar to that of Hu et al. [85], in which they compare metamorphic testing and runtime assertion checking. However, their study only compares deterministic applications. To our knowledge, we are the first to compare

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

Gotleib and Botella coined the term "automated metamorphic testing" [68] to describe how the process can be conducted automatically, but their work focuses on the automatic creation of input data that would reveal violations of metamorphic properties, and not on automatically checking that those properties hold after execution. That is, for specified metamorphic property of a given function, they attempt to construct test data that would violate that property. Like most static techniques, this approach has issues related to scalability, as it only is useful in practice for individual functions and not for entire applications; additionally, the implementation is limited in scope because of necessary assumptions related to the use of constraint logic programming [118], which cannot handle arbitrary data structures and pointers.

Beydeda [18] first brought up the notion of combining metamorphic testing and self-testing components so that an application can be tested at runtime, as we do in Metamorphic Runtime Checking, but did not investigate an implementation or consider the effectiveness on testing applications without oracles. 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 [19]. Our work extends that initial idea by providing implementation details and evidence of feasibility and effectiveness.

Metamorphic Runtime Checking is in a sense similar to fuzz testing [177] in that function inputs are modified during program execution, and then the function is re-run with the new inputs. However, in fuzz testing, there is no expected relationship between the original output and the output that comes from the modified input, unlike in metamorphic testing, in which there is an expected relationship between the outputs, and a violation of that relationship is indicative of a defect. In fuzz testing, the goal is to force erroneous behavior (e.g., a program crash), which is more appropriate for detecting security vulnerabilities than for the types of defects investigated in this thesis.

Outside of Guderlei and Mayer's work on statistical metamorphic testing [71], we are not aware of any other investigation of using metamorphic testing techniques for testing non-deterministic applications. As noted in the description of Heuristic Metamorphic Testing (Section 3.3), statistical metamorphic testing can only be used for applications that produce outputs that have statistical properties, such as mean and variance, whereas Heuristic Metamorphic Testing is applicable to the more general case of non-deterministic applications in which profiles of outputs can expected to be "similar" (according to some domain-specific definition of similarity) across multiple executions.

6.3 Self-Testing Software

While the notion of "self-checking software" is by no means new [197], much of the recent work in self-testing components has focused on commercial off-the-shelf (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 [104] to record testing and execution

history and make the information available to a software tester, and "just-in-time testing" [105] to check component compatibility with client software at the time of its use. Work in "built-in-testing" [188] has included investigation of how to make components testable [17, 20, 22, 117], and frameworks for executing the tests [53, 116, 123], including those in Java programs [54], or through the use of aspect-oriented programming [115]. However, none of these address the issue of testing applications without test oracles, or of using properties of the individual functions to perform system testing.

In light of all these important contributions, In Vivo Testing differentiates itself by providing the ability to test any part of the system without requiring extensive modification to the original source to provide special functional and testing interfaces [9, 185], or enforcing a rearchitecture of the application to allow for the use of testers and controllers/handlers [14, 126, 185]. The advantage of the In Vivo Testing approach over these others is that we are providing a framework for allowing an application to test itself with minimal or no modification, as opposed to prescribing a methodology for developing an application so that it may be tested after its deployment. However, it may be interesting to consider how an application should be designed in order to maximize the effectiveness of In Vivo Testing; this is described in Future Work (Section 7.2).

6.4 Runtime Testing

6.4.1 Testing in the Deployment Environment

All of the testing approaches presented in this thesis allow for the software to test itself, conceivably after the software has been deployed into the field. This is inspired in part by the idea of "perpetual testing" [148, 158, 199], 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 these 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"). The In Vivo Testing approach is a type of perpetual testing in which the tests are executed from within the context of the running application and do not alter the application state from the user's perspective.

In Vivo Testing is also a form of "residual testing" [149]. 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 [137, 149] or by comparing the coverage to specifications [136]. However, this work does not consider the actual execution of tests in the deployment environment, as we explore here. Those approaches describe measurements of the residue, whereas we are attempting to discover the residual defects by conducting tests. The In Vivo Testing 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 defects could reside in already-tested code.

Other approaches to testing software in the field include the monitoring, analysis, and profiling of deployed software, as surveyed by Elbaum and Hardojo [56]. These include the following:

- **Skoll** [94, 121] has extended the idea of round-the-clock "continuous testing" [162] 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, In Vivo 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, In Vivo Testing executes tests as the application runs in its deployment environment. Additionally, although the In Vivo approach does not currently address performance testing, as Skoll does, our approach could be enhanced to maintain records of resource utilization of the individual units tested, for instance to help detect bottlenecks where optimization may be warranted, or in cases where *a priori* assumptions about resource utilization turn out to be off base in the field for a particular installation.

- **Gamma** [144, 145] uses an approach called "software tomography" to divide monitoring tasks and then reassemble gathered information. Unlike In Vivo testing, Gamma does not seek to actively test the application as it runs in the field. Rather, it only monitors the program, typically considering coverage (statement, basic block, method, or class) for purposes of profiling. This information can then be used for onsite modification of the code (for instance, by distributing a patch) to fix defects, but the approach does not detect defects on its own. It has, however, been used to perform anomaly detection [11], though this only considers program execution paths and not whether tests pass or fail when run in the field. It is possible, then, that the approaches described in this thesis could be used in conjunction with Gamma, which could dynamically modify the profiling tasks at runtime according to the results of In Vivo Tests or Metamorphic Runtime Checking.

- **Cooperative Bug Isolation** [101] 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 defects. This approach is more suited towards fault localization than fault detection, as it requires some other mechanism to indicate that a particular program execution is faulty, so that it can then be debugged using data

collected from non-faulty executions. The work on Cooperative Bug Isolation to date has considered cases in which the fault cause the program to crash or hang. For faults related to functional or computational correctness, however, In Vivo tests could conceivably be used to indicate that there is a defect.

6.4.2 Reducing Performance Overhead

In Section 5.6.5, we described how it is possible to reduce the overhead of runtime monitoring and testing by only conducting tests in previously-unseen states. Other approaches to reducing this overhead have included the use of static analysis to remove unnecessary instrumentation [198], or pre-determining when to execute uninstrumented "fast cases" instead of instrumented "slow cases" [101]. The techniques could possibly be combined to reduce performance costs even further.

Much of the work in the representation of application state at runtime has focused on anomaly detection, i.e., determining that the application is in a state that is outside the range of what is expected [75]. These works also deal with the issue of "has this state been seen before?", but the representation of state in those approaches is based on a finite state machine that considers the execution path up to that point, and not the set of variable values. However, future work could investigate how state-based anomaly detection techniques and the approach presented here could be combined, for instance by further simplifying the representation of expected states according to semantic equivalence.

6.5 Domain-Specific Testing

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

6.5.1 Machine Learning

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular [23, 30, 201], 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 [51] and Weka [194] are two of several frameworks that aid machine learning developers, but the testing functionality they provide is focused on comparing the quality of the results, and not evaluating the "correctness" of the implementations. Repositories of "reusable" data sets have been collected (e.g., the UCI Machine Learning Repository [138]) for the purpose of comparing result quality, i.e., how accurately the algorithms predict, but not for the software engineering sense of testing.

Testing of intrusion detection systems [119, 139], intrusion tolerant systems [113], and other security systems [13] has typically addressed quantitative measurements like overhead, false alarm rates, or ability to detect zero-day attacks, but does not seek to ensure that the implementation is free of defects, as we do here. An intrusion detection system with very few or no false alarms could still have bugs that prevent it from detecting many (or any) actual intrusions, making it completely undependable.

6.5.2 Simulation and Optimization

Researchers concerned with the verification of simulation software often acknowledge the need for software testing, but typically do not present techniques beyond what is common for all types of software [12, 92, 166], such as test-driven development or using code reviews. Others have focused on using formal specifications [180], as have researchers investigating the verification of optimization software, as in compiler optimizations [96, 99]. However, as described above, the use of formal languages to act as an oracle can be challenging from a practical point of view, given that the specification often needs to be complete in order to be useful. Additionally, the creation of a formal specification can

be fairly complex after the software has already been developed, and requires intimate knowledge of the algorithm being implemented. In the studies presented in Section 3.4, however, we showed that even with a basic understanding of the simulation (JSim) and optimization (gaffitter) software, we were able to create metamorphic properties that were more effective than other approaches at finding defects.

6.5.3 Scientific Computing

Some of the recent research presented at the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering addressed many of the issues that arise in testing applications in domains without test oracles. Hannay et al. pointed out that scientific computing software is often developed by scientists (as opposed to professional programmers or software engineers) who understand that software testing is important, but generally feel that they have insufficient understanding of software testing concepts [77]. Heroux and Willenbring suggested a test-driven approach for developing such software [81]; as pointed out in Section 2.4, the identification of metamorphic properties early in the specification phase would likely enhance the effectiveness of metamorphic testing. Hook and Kelly described a testing approach for scientific software that, aside from algorithm verification and scientific validation, calls for "code scrutinization", i.e., specifically looking for common defects such as off-by-one errors or incorrect use of array indices [84]. As demonstrated by our empirical studies, metamorphic testing may be used instead of code inspection to effectively detect such errors instead.

Last, although we did not apply any of the metamorphic testing techniques to applications in scientific computing per se, we note that many such applications depend on machine learning components, which we have investigated here. Also, Chen et al. [33] have applied metamorphic testing to scientific computing programs (specifically in bioinformatics), with favorable results.

Chapter 7

Conclusion

7.1 Contributions

In this thesis, we have presented new techniques for applying metamorphic testing to software without test oracles, evaluated those techniques, and demonstrated that the techniques can also be applied to testing any type of application as it runs in the deployment environment. The main contributions of this thesis are as follows:

- We are the first to present a general set of **metamorphic testing guidelines** (Section 2.4) that can be followed to assist in the formulation and specification of metamorphic properties. We have demonstrated that numerous applications in domains without test oracles exhibit metamorphic properties that fall into the seven categories that we identified, based generally on linear transformations, translations, or set modification. We have also provided suggestions on how metamorphic testing can fit into the overall software development process.

- We have introduced a new testing technique called **Heuristic Metamorphic Testing** (Section 3.3), which is specifically designed to test non-deterministic applications that do not have test oracles. In this approach, the application is run multiple times to create a profile of the expected outputs, according to a domain-specific heuristic, and

then the metamorphic transformation is applied to the input. The application is then run multiple times again with the new input, and a measurement is taken to see if the profile of the new outputs is statistically similar to what is expected. If not, then a defect has been detected.

- We have presented the results of new **empirical studies** (Section 3.4) showing that the metamorphic testing approaches are more effective at detecting defects in applications without test oracles than other software testing techniques, including the use of partial oracles and runtime assertion checking. We are the first to conduct such studies on non-deterministic applications.

- We have described a new type of testing called **Metamorphic Runtime Checking** (Section 4.1). Rather than test the application using system-level metamorphic properties, this approach involves the execution of function-level metamorphic tests from within the running application, i.e., the application launches its own tests at designated points in the program, within its current runtime context and using the current function inputs. We have demonstrated that this approach can detect defects not found by system-level metamorphic testing alone, and is more effective overall at detecting defects in functions for which metamorphic properties have been identified.

- Last, we have presented a new technique called **In Vivo Testing** (Section 5.1). This approach allows software to perform any type of test (not just metamorphic tests) as defined by the programmer, including unit or integration tests, or special "In Vivo tests" that are designed to check properties of the system that should hold in any application state. The In Vivo Testing framework, called Invite, ensures that any side effects from the tests do not affect the state of the application from the end users' perspective. We have also demonstrated that our implementation is feasible for use in the deployment environment, in that hundreds or even thousands of tests can be run with only one second of performance overhead.

This work has also led to the publication of four conference papers on metamorphic testing [131, 132, 133, 195], and four conference and workshop papers on In Vivo Testing [40, 49, 130, 134]. Additionally, six graduate students and three undergraduates gained research experience by participating in the work described in this thesis.

7.2 Future Work

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

7.2.1 Immediate Future Work Possibilities

- **Ensure that the test code does not modify anything external to the system.** The most critical limitation of the current Amsterdam, Columbus, and Invite implementations is that anything external to the application process itself, e.g., database tables, network I/O, etc., is not included in the sandbox and modifications made by a test may therefore affect the external state of the original application. Although this appears to limit the usefulness of the approach, we note, however, that in our experiments, the current sandbox implementation (which provides the test process with its own memory space and own view of the file system) was sufficient for the applications we tested: none of the applications used an external database or network I/O (the network intrusion detection system PAYL has an offline mode that was used in our experiments). For database-driven applications, it may be possible to automate the creation of sandboxed database tables using copy-on-write technology (as in Microsoft SQL Server [1]) or "safe" test case selection techniques that ensure that there will be no permanent changes to the database state as a result of the tests [191, 192].

- **Support automatic fault localization.** Although the approaches can record a test

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

failure and know which test failed, it may not be obvious what is the root cause of the failure (invalid system state, invalid function arguments, configuration, environment, combination thereof, etc.). Thus, the system could take a snapshot of the relevant parts of the state (i.e., the ones that could have affected the outcome of the test) and record those in the failure log as well, for further analysis. If these logs are aggregated by the software developer, a failure analysis technique (e.g., [11] or [101]) could be used to try to isolate the fault. However, given that research into fault localization techniques it typically targeted at defects that cause the application to crash, or assume the presence of an oracle, challenges will arise in determining the source of the error in applications for which the correct output cannot be known in advance.

- **Automatically detect properties that can be used to aid in the generation of tests.** Although prior work has been done in automatically determining algebraic specifications [80] and in categorizing metamorphic properties [131], as of now it is necessary for the software developer or tester to discover and specify the properties and/or write the tests required for metamorphic testing. As discussed in Section 2.4, it may be possible to automate this process, using static or dynamic techniques.

7.2.2 Possibilities for Long-Term Future Directions

- **Explore soundness of metamorphic properties.** The violation of a metamorphic property does not *necessarily* indicate a defect: it is possible that the property may not be sound. Future work could investigate how a tester would be able to tell the difference. On a related note, 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 [72]. For instance, we previously pointed out a metamorphic property in the ML ranking algorithm MartiRank that permuting the order of the

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

- **Multi-process or distributed applications.** As multi-core processing and cloud computing become more and more prevalent, many applications in domains like scientific computing use parallel and/or distributed processing techniques to reduce execution time. This raises new challenges of testing these applications, particularly when using approaches based on checking properties of the code in various application states (such as Metamorphic Runtime Checking or In Vivo Testing), since those states may be distributed across multiple processes and/or multiple machines. Static techniques have been suggested for testing such applications [170], but run into issues of scale and the need for a representative model to act as an oracle. Future work could consider ways of checking properties (metamorphic or otherwise) in these applications when the properties rely on data that may not all be in a single process.

- **Enable collaborative defect detection and notification.** Aside from just sharing the performance load of conducting the In Vivo tests, the frameworks could be modified to allow instances in so-called "application communities" [108] 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 application domains, for instance scientific computing, 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 correct. Additionally, the distributed In Vivo Testing approach (Section 5.3.5)

could be modified so that the server dynamically reassigns testing responsibilities based on global testing load or global overhead, e.g., ensuring that the overhead across all instances is minimized. Whereas other systems, such as Skoll [121], do this statically by using an *a priori* planning algorithm, testing responsibilities could be dynamically assigned based on each instance's execution profile, so as to increase the likelihood of running tests in a variety of different environments, configurations, and application states.

- **More efficient test execution.** Future work could also consider which functions to instrument so as to increase the probability of detecting defects, the percentage of function calls that should launch tests, the optimal timing for when tests should be run, or how to test code that is not in the execution path, since the current frameworks only execute tests based on actual invocations of the instrumented functions. This would most likely vary greatly depending on the type of application and the defects that are being targeted.

- **Evaluate impact on the software development process.** As described previously in Section 2.4, in practice we would expect that the metamorphic properties would be identified as part of the planning and architecture phase, and included in the program specification. However, the fact that metamorphic testing (at either the function level or the system level) will be used may affect the design of the application, e.g., creating smaller functions that are easier to test, or that take inputs and produce outputs for which it is easy to check the metamorphic properties. Some application designs may be more amenable to metamorphic testing than others. Moreover, we can consider the way that software is developed in the domains of interest (scientific computing, machine learning, etc.) and get an understanding of how metamorphic testing can be integrated into the processes used in those fields.

7.3 Conclusion

In this thesis we have explored 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. We have demonstrated that such approaches are feasible for revealing defects in real world programs, and are more effective than using other 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 [16]. As programs without test oracles - such as those in the domains of machine learning, simulation, optimization, and scientific computing - become more and more prevalent and mission-critical, ensuring their quality and reliability gains the utmost importance.

Chapter 8

Bibliography

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

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice Hall, 1981.

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

[4] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, pages 105–114, 2000.

[5] Apache Java Caching Solution (JCS). http://jakarta.apache.org/jcs/.

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

[7] Apache Tomcat. http://tomcat.apache.org/.

[8] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. In *Proc. of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002)*, pages 16–18, 2001.

[9] 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.

[10] D. A. Augusto. Genetic algorithm file fitter. http://gaffitter.sourceforge.net/.

[11] 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.

[12] O. Balci. Verification, validation and accreditation of simulation models. In *Proc. of the 29th conference on winter simulation*, pages 135–141, 1997.

[13] 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 *Proc. of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[14] 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.

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

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

[17] 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.

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

[19] S. Beydeda. Personal communication, 2008.

[20] 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.

[21] J. Biskup. *Security in computing systems challenges, approaches, and solutions*. Springer-Verlag, 2009.

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

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

[24] S. Brilliant, J. C. Knight, and N. G. Leveson. The consistent comparison problem in n-version software. *IEEE Transactions on Software Engineering*, 15:1481–1485, 1989.

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

[26] J. Carver. Report from the second international workshop on software engineering for computational science and engineering. *Computing in Science & Engineering*, 11(6):14–19, 2009.

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

[28] 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.

[29] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/˜cjlin/libsvm.

[30] 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.

[31] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of the 8th Symposium on Fault-Tolerance Computing*, pages 3–9, 1978.

[32] 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.

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

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

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

[36] 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.

[37] 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.

[38] 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[39] Y. Cheon. Abstraction in assertion-based test oracles. In *Proc. of Seventh International Conference on Quality Software*, pages 410–414, 2007.

[40] 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 (ICST)*, pages 509–512, April 2008.

[41] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg: a tool for generating symbolic test programs and oracles from operational specifications. In *Proc. of the 8th European software engineering conference*, pages 301–302, 2001.

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

[43] 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.

[44] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proc. of the 29th International Conference on Software Engineering (ICSE)*, pages 261–270, 2007.

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

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

[47] Hewlett-Packard Company. *Programming with Judy*. 2001.

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

[49] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing for software vulnerability detection. In *Proc. of the Fourth International Workshop on Secure Software Engineering (SecSE)*, pages 525–530, 2010.

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

[51] 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.

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

[53] G. Denaro, L. Mariani, and M. Pezzè. 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.

[54] 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.

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

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

[57] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, pages 57–72, 2001.

[58] 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.

[59] E. Eskin, W. Lee, and S. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *Proc. of DARPA Information Survivabilty Conference and Exposition II (DISCEX)*, 2001.

[60] G. W. Evans, T. B. Gor, and E. Unger. A simulation model for evaluating personnel schedules in a hospital emergency department. In *Proc. of the 28th Conference on Winter Simulation*, pages 1205–1209, 1996.

[61] G. D. Everett and R. McLeod Jr. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Press, 2007.

[62] R. E. Fairley. Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, April 1978.

[63] A. Freedman. *Computer Desktop Encyclopedia*. AMACOM, 1999.

[64] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[65] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*, pages 321–330, 2008.

[66] J. D. Gannon, P. McMullin, and R. G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[67] W. S. Gosset. The probable error of a mean. *Biometrika*, 6(1):1–25, March 1908.

[68] A. Gotleib and B. Botella. Automated metamorphic testing. In *Proc. of 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 34–40, 2003.

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

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

[71] R. Guderlei and J. Mayer. Statistical metamorphic testing - testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proc. of the Seventh International Conference on Quality Software*, pages 404–409, 2007.

[72] 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 (PLDI)*, pages 69–82, 2002.

[73] D. Hamlet. Random testing. *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[74] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2006.

[75] 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.

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

[77] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009.

[78] L. Hatton. The chimera of software quality. *Computer*, 40:102–103, 2007.

[79] L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20:786–797, 1994.

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

[81] M. A. Heroux and J. M. Willenbring. Barely sufficient software engineering: 10 practices to improve your CSE software. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 15–21, 2009.

[82] J. W. K. Ho, M. W. Lin, S. Adelstein, and C. G. dos Remedios. Customising an antibody leukocyte capture microarray for systemic lupus erythematosus: Beyond biomarker discovery. *Proteomics - Clinic Application*, in press, 2010.

[83] D. Hoffman. Heuristic test oracles. *Software Testing and Quality Engineering*, pages 29–32, 1999.

[84] D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 59–64, 2009.

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

[86] P. Inella and O. McMillan. An introduction to intrusion detection systems. Tetrad Digital Integrity, LLC, 2001.

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

[88] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proc of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[89] JUnit. http://www.junit.org/.

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

[91] N. Keetha, L. Wu, G. Kaiser, and J. Yang. Distributed eXplode: A high-performance model checking engine to scale up state-space coverage. Technical Report CUCS-051-08, Department of Computer Science, Columbia University, 2008.

[92] J. P. C. Kleijnen. Verification and validation of simulation models. *European Journal of Operational Research*, 82(1):145–162, April 1995.

[93] 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.

[94] 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.

[95] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[96] D. Lacey, N. D. Jones, E. van Wyk, and C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 283–294, 2002.

[97] 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.

[98] 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.

[99] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of the ACM SIGPLAN conference on programming language design and implementation*, pages 220–231, 2003.

[100] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[101] 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.

[102] H. W. Lilliefors. On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318):399–402, June 1967.

[103] R. J. Lipton. New directions in testing. *Distributed Computing and Cryptography*, 2:191–202, 1991.

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

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

[106] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proc. of the 10th European Software Engineering Conference*, pages 286–295, 2005.

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

[108] 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.

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

[110] 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 (ASPLOS)*, pages 37–48, 2006.

[111] 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.

[112] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[113] 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.

[114] P. C. Mahalanobis. On the generalised distance in statistics. *Proceedings of the National Institute of Science of India*, 12:49–55, 1936.

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

[116] 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.

[117] L. Mariani, M. Pezzè, 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.

[118] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

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

[120] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proc. of the 18th International Conference on Automated Software Engineering (ASE)*, pages 164–173, 2003.

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

[122] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proc. of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE)*, 2000.

[123] M. Merdes, R. Malaka, D. Suliman, B. Paech, D. Brenner, and C. Atkinson. 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.

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

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

[126] 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.

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

[128] C. Murphy and G. Kaiser. Empirical evaluation of approaches to testing applications without test oracles. Technical Report CUCS-039-09, Dept. of Computer Science, Columbia Univ., 2009.

[129] C. Murphy, G. Kaiser, and M. Arias. Parameterizing random test data according to equivalence classes. In *Proc. of the 2nd International Workshop on Random Testing*, pages 38–41, 2007.

[130] C. Murphy, G. Kaiser, M. Chu, and I. Vo. Quality assurance of software applications using the in vivo testing approach. In *Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 111–120, 2009.

[131] 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.

[132] C. Murphy, K. Shen, and G. Kaiser. Automated metamorphic system testing. In *Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA)*, pages 189–199, 2009.

[133] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 436–445, 2009.

[134] C. Murphy, M. Vaughan, W. Ilahi, and G. Kaiser. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proc. of the 5th International Workshop on Automation of Software Test (AST)*, 2010.

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

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

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

[138] 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.

[139] 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.

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

[141] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. of the 10th International Symposium on the Foundations of Software Engineering (FSE)*, pages 11–20, 2002.

[142] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, March 1991.

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

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

[145] A. Orso, D. Liang, and M. J. Harrold. Gamma system: Continuous evolution of software after deployment. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 65–69, 2002.

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

[147] 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.

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

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

[150] D. K. Peters and Parnas D. L. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

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

[152] M. Pidd. *Computer simulation in management science*. Wiley, 1998.

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

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

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

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

[157] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of netnews. In *Proc. of the 1994 ACM conference on computer supported cooperative work (CSCW)*, pages 175–186, 1994.

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

[159] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. of the 14th International Conference on Software Engineering (ICSE)*, pages 105–118, 1992.

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

[161] H. L. Royden. *Real Analysis*. Prentice Hall, 1988.

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

[163] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

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

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

[166] R. G. Sargent. Verification and validation of simulation models. In *Proc. of the 37th conference on winter simulation*, pages 130–143, 2005.

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

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

[169] S. Sidiroglou, O. Laadan, N. Viennot, C.-R. Pérez, A. D. Keromytis, and J. Nieh. Assure: Automatic software self-healing using rescue points. In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[170] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proc. of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, pages 157–167, 2006.

[171] Arena Simulation Software. http://www.arenasimulation.com/.

[172] Y. Song. Personal communication, 2009.

[173] C. Spearman. Footrule for measuring correlation. *British Journal of Psychology*, 2:89–108, June 1906.

[174] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Proc. of the 18th International Symposium on Software Reliability*, pages 117–126, 2007.

[175] StackSafe, Inc. IT Operations Research Report: Testing Maturity. Technical report, 2008.

[176] S. Stolfo. Personal communication, 2008.

[177] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

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

[179] N. Tillman and W. Schulte. Parameterized unit tests. In *Proc. of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[180] W. T. Tsai, X. Liu, Y. Chen, and R. Paul. Simulation verification and validation by dynamic policy enforcement. In *Proc. of the 38th annual Symposium on Simulation*, pages 91–98, 2005.

[181] C. D. Turner and D.J. Robson. State based testing and inheritance. Technical Report TR-1/93, Durham, GB, 1993.

[182] E. Tyugu and A. Saabas. Problems of visual specification languages. In *Proc. of Information Technologies in Science, Education, Telecommunication and Business*, pages 155–157, 2003.

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

[184] G. Viguier-Just. Personal communication, 2009.

[185] 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.

[186] 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.

[187] X. Wang, J. Wang, and Z.-C. Qi. Automatic generation of run-time test oracles for distributed real-time systems. *Lecture Notes in Computer Science*, 3235/2004:199–212, 2004.

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

[189] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proc. of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

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

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

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

[193] A. Wise. JSim agent behavior specification language.

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

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

[196] D. J. Yantzi and J. H. Andrews. Industrial evaluation of a log file analysis methodology. In *Proc. of the 5th International Workshop on Dynamic Analysis*, 2007.

[197] 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.

[198] S. H. Yong and S. Horwitz. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design*, 27(3):313–334, November 2005.

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

[200] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. of the 10th ACM SIGSOFT symposium on foundations of software engineering*, pages 1–10, 2002.

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

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

[203] Z. Q. Zhou, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of web search engines in the absence of an oracle. Technical Report TR-2007-06, Dept. of Computer Science, Hong Kong University, 2007.