Using JML Runtime Assertion Checking to Automate Metamorphic Testing in Applications without Test Oracles

Christian Murphy, Kuang Shen, Gail Kaiser
Department of Computer Science, Columbia University, New York NY 10027
{cmurphy, ks2555, kaiser}@cs.columbia.edu

Abstract

It is challenging to test applications and functions for which the correct output for arbitrary input cannot be known in advance, e.g. *some computational science or machine learning applications. In the absence of a test oracle, one approach to testing these applications is to use metamorphic testing: existing test case input is modified to produce new test cases in such a manner that, when given the new input, the application should produce an output that can be easily be computed based on the original output. That is, if input* x *produces output* f(x)*, then we create input* x' *such that we can predict* f(x') *based on* f(x)*; if the application or function does not produce the expected output, then a defect must exist, and either* f(x) *or* f(x') *(or both) is wrong. By using metamorphic testing, we are able to provide built-in "pseudo-oracles" for these so-called "non-testable programs" that have no test oracles.*

In this paper, we describe an approach in which a function's metamorphic properties are specified using an extension to the Java Modeling Language (JML), a behavioral interface specification language that is used to support the "design by contract" paradigm in Java applications. Our implementation, called Corduroy*, pre-processes these specifications and generates test code that can be executed using JML runtime assertion checking, for ensuring that the specifications hold during program execution. In addition to presenting our approach and implementation, we also describe our findings from case studies in which we apply our technique to applications without test oracles.*

1. Introduction

Assuring the quality of applications such as those in the fields of scientific calculations, optimizations, data mining, machine learning, *etc.* presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [40]. These applications fall into a category of software that Weyuker describes as *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known"* [40].

One approach to testing such applications is to use a "pseudo-oracle" [16], in which multiple implementations of an algorithm process an input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [28].

In the absence of multiple implementations, however, metamorphic testing [9] can be used to produce a similar effect. Metamorphic testing is designed as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure, in order to try to find uncovered flaws. Instead of being an approach for test case selection, it is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. In metamorphic testing, if input x produces an output $f(x)$, the function's 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 output $f(t(x))$, based on the (already known) value of $f(x)$. If the output is not as expected, then a defect must exist. Of course, this can only show the existence of defects and cannot demonstrate their absence, since the correct output cannot be known in advance (and even if the outputs are as expected, both could be incorrect), but metamorphic testing provides a powerful technique to reveal defects in such non-testable programs by use of a built-in pseudo-oracle.

To automate this process in a general-purpose implementation, it is necessary to specify the metamorphic properties so that a test engine can generate the test case inputs and know how to compare the outputs. Here, we present an extension to the Java Modeling Language (JML) [29], a behavioral interface specification language that is used to support the "design by contract" paradigm in Java applications. In JML, an application designer or developer can formally specify a module's behavior by adding annotation comments within the Java source code, such as its assumptions (preconditions) or guarantees (postconditions), as well as any invariant properties. Numerous JML tools support "runtime assertion checking" [6] for ensuring that the specifications hold during program execution. However, to date none address the particular needs of functions that do not have a reliable test oracle, and JML itself does not support the mechanisms needed for metamorphic testing. Checking that the functions meet their preconditions and postconditions is not sufficient to demonstrate that the return value of the function is correct (or, rather, not incorrect) for the particular input, especially given that the correct output cannot be known in advance, and thus cannot be specified as part of the postcondition in terms of the input or precondition.

This paper makes three contributions: (1) a testing approach that uses a variant of metamorphic testing to perform quality assurance of functions in applications that do not have test oracles; (2) an extension to the JML specification language that allows for the specification of a function's metamorphic properties; and (3) an implementation framework called *Corduroy*, which converts the specification of metamorphic properties into test methods, which can then be executed using JML runtime assertion checking (see Figure 1). In addition to presenting our approach and implementation, we also describe our findings from case studies in which we apply our technique to non-testable programs.

2. Background

We have previously explored an approach in which input was manually manipulated to perform metamorphic testing of applications without test oracles [31]. Here, we improve on that work in two ways. First, we refine the approach to perform metamorphic testing of individual functions, rather than of the application as a whole, to allow for the investigation of more metamorphic properties (and thus more test cases) and better fault localization. Second, rather than using one-off, ad-hoc scripts to modify the input data for metamorphic testing and to compare the outputs, we introduce an extension to the Java Modeling Language (JML) [29] to specify the functions' metamorphic properties. These extensions, combined with the automatic generation of test code and translation into plain JML that invokes that test code, allow us to combine JML's powerful specification capabilities with its "runtime assertion checking" [6] feature to check that the properties hold when provided *any* test input, which is assumed to previously exist.[1] If the specifications are not met, then a defect has been detected.

A simple example - for exposition purposes only - of a function to which metamorphic testing could be applied would be one that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result. For instance, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the devation from the mean would still be the same (think about the values being "flipped" around the origin on the number line).

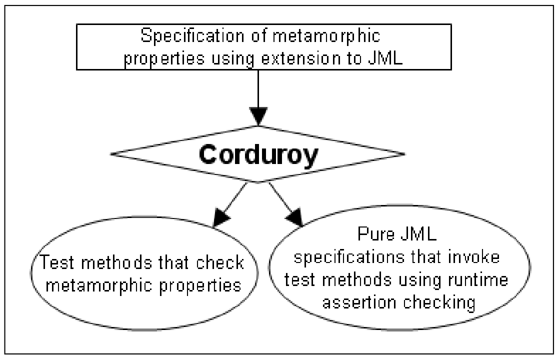
Specification of metamorphic properties using extension to JML
Corduroy
Test methods that check metamorphic properties
Pure JML specifications that invoke test methods using runtime assertion checking

Figure 1. Corduroy converts metamorphic properties into test methods and JML specifications that can be checked at runtime

Furthermore, we know that there are other transformations that will alter the output, but in a predictable way. For instance, if each value in the set is multipled by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers, we can 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 result of only the first test case, we can predict what the other three should be.

Metamorphic testing generally would not be needed for this trivial example, but clearly can be very useful in the absence of an oracle: regardless of the values in the data set, and even if the correct output of an application or function could not be known in advance, if the outputs are not as expected, then there must be a defect in the implementation of the function. Although the use of these simple identities for testing numerical functions is not unique to metamorphic testing [14], the approach can be used on a broader domain of any functions that display metamorphic properties.

[1] This could be real-world user input, but we assume for the purposes of this paper that the initial input is produced somehow in the lab.

Machine learning applications are a good example of non-testable programs that can be tested using this approach. For instance, 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 [39]. 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 in this case that is entirely dependent on the model. However, if we take the new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect must exist.

3. Related Work

Applying metamorphic testing to situations in which there is no test oracle has previously been studied by Chen *et al.* [10]. In some cases, these works have looked at situations in which there cannot be an oracle for a particular application [11]; in others, the work has considered the case in which the oracle is simply absent or difficult to implement [8]. However, this previous work has mostly looked at system-level testing as opposed to internal unit- and integration-level testing as we present here. Additionally, their work did not use any notation for specifying the metamorphic properties, but rather relied on a tester to manually perform the transformations; in this work, we automate those transformations using a specification language.

Other formal specification languages, such as Alloy [25], ANNA [30], Larch [21], Z [2], *etc.* could also be used as starting points for the specification of metamorphic properties. We have chosen JML as an implementation vehicle because it was already familiar to the authors, though in principle our approach could apply to any analogous assertion checking system.

Metamorphic properties are similar in some ways to algebraic specifications [14], though algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (*e.g. pop(push(X))* == *X* in a Stack), but do not describe how an arbitrary function should react when its input is changed. The runtime checking of algebraic specifications has been explored in [33] and [35], though neither work considered the specification of metamorphic properties, and the particular issues that arise from testing without oracles. Others have looked at the automatic detection of algebraic specifications, in particular [24], and of program invariants in general (*e.g.* DIDUCE [23], Daikon [17], Houdini [18], *etc.*). Even in the cases in which program invariants, algebraic specifications, formal specification languages are used to act as oracles (assuming they are complete, which may be an undecidable problem [36]), work to date has focused primarily on consistency checking of abstract data types [36] and has not sought to create oracles for applications and functions that do not otherwise have them.

Previous work on using JML for runtime assertion checking includes Cheon and Leavens' initial description of JML [12] and the JML4 extension to Eclipse [7], as well as tools like JAJML for handling loop annotations [22], DSD-Crasher for creating test cases based on invariant detection and both dynamic and static analysis [15], and JMLAutoTest for automatically generating and executing unit tests from JML specifications [42]. Our work adds to and complements this list by presenting an approach that is focused on metamorphic testing and applied in particular to programs without test oracles.

Last, Beydeda [5] identified the notion of combining metamorphic testing and self-testing components so that an application's metamorphic properties can be tested automatically, but did not present an implementation or use any particular specification language. And Kim *et al.* investigated a runtime testing technique that depended on mutation testing [27], which is superficially similar to our approach (in which we "mutate" the input, though, not the code), but they did not look at non-testable programs.

4. Approach

The approach we describe here is a variant of metamorphic testing in which the metamorphic properties of a function are specified using an extension to JML, converted to test code, and then checked as the program runs on test input data. Although this approach could conceivably be applied to any type of application or function, it is most useful for those without a test oracle.

For instance, as in the standard deviation example described above in Section 2, whenever the function is called, its argument can be passed along to a test method, which will multiply each element in the array by -1 and check that the two calculated output values are equal. This does not require a test oracle for the particular input; the metamorphic relationship specifies its own test oracle. It is true that if the two outputs are equal, they are not necessarily correct, but if they are not equal, then a defect must exist.

Our testing approach entails three steps:

1. Specify metamorphic properties. For each function to be tested, the software developer specifies its metamorphic properties using our extension to the JML specification language, described in Section 5. These specifications can either be placed directly into the code, or into separate files, as permitted by JML.

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

Table 1. Classes of metamorphic properties

In [31] we enumerated six classes of metamorphic properties, as summarized in Table 1, which can be used as guidelines for the specifications to be written in JML; although these represent only a sample set that were particular to the domain of machine learning applications, and other classes may exist, they should also hold for applications in other domains, and we have used those classes to guide the types of metamorphic properties that can be expressed in our extension to JML.

2. Convert the specifications into tests. In this step, the developer uses our tool, called *Corduroy*, to process the specifications and convert them into test methods. These test methods will be added to the original source code, as described in Section 6. Additionally, JML post-conditions will be added to each function's specification so that the tests can be executed and the results can be compared.

3. Compile and execute the code. Using any JML-compliant tool, such as [7] or [12], the developer can then compile the code and execute it with its regular test cases. If the JML virtual machine supports runtime assertion checking and it is enabled, the functions' post-condition checks will invoke the test methods generated in Step 2. If a test fails, then a defect has been detected.

5. Extensions to JML Syntax

Our approach extends the JML syntax to allow for the specification of metamorphic properties. As in JML, the properties are specified in annotations in the comments preceding the method with which they are associated, or in a separate file. In our extension, the metamorphic properties are specified in a line starting with the tag "@meta" and then are followed by a Java boolean expression that states the property.

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

Figure 2. Example of specification of metamorphic properties for sine using JML

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

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

5.1. Comparing values

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

5.2. 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 [31], we have also added special keywords to the JML syntax, using the JML style of starting keywords and operators with a backslash. These allow for the execution of operations on arrays (of Objects or primitives) or on classes that implement the Java Collection interface that would used during the test; Table 2 explains these built-in functions.

An example of the use of these keywords appears in Figure 3. When calculating the standard deviation for an array of integers, shuffling the values should not affect the result, since the calculation does not depend on the initial ordering of the elements. However, multiplying each element by 2 is expected to double the calculated standard deviation.

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

Table 2. Additional keywords added to JML for manipulating arrays

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

Figure 3. Example of using built-in array functions for specifying metamorphic properties

5.3. 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 specifiying metamorphic properties, using if/else notation as opposed to the question mark-colon notation currently supported in JML. Figure 4 shows an example.

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

Figure 4. Conditional metamorphic property

5.4. Handling non-determinism

Functions that are non-deterministic may still have metamorphic properties, though these would be 1-to-many relationships of inputs to possible outputs, rather than 1-to-1 mappings as we have discussed so far. For instance, a function that solves a quadratic equation may return the two possible values in an array, where either [x_1, x_2] or [x_2, x_1] is correct. Thus, the metamorphic property would need to check that the new output is equal to one of these two possibilities. In other cases, the new output might be expected to fall within some range of numbers. To make these properties easier to express, we add two additional boolean functions, as described in Table 3.

Consider, for example, a function in a personal finance application that simulates market conditions and predicts the value of the user's portfolio after a certain amount of time. This function may use a Monte Carlo algorithm that uses randomness to simulate many possibilities and then reports the average as its output. If the value of each holding in the portfolio is doubled, we cannot expect that the predicted value will exactly be doubled, since the function is non-deterministic, but we may be able to specify that the value should not be *less* than the original output, and perhaps should not be more than four times that value. Thus, we can specify this metamorphic property as demonstrated in Figure 5.

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

Table 3. Additional keywords for handling non-determinism in specifications

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

Figure 5. Example of metamorphic properties specifying a range of values

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

6. Implementation

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

Note that after Corduroy has pre-processed the code, runtime assertion checking can easily be disabled in the JML runtime environment, so that recompilation is not necessary; Corduroy does not "force" any runtime checking, it only enables it. However, as Corduroy is not a compiler, but rather relies on the JML compiler tools, it does assume that the application code is free of syntax and semantic errors before it creates the metamorphic tests.

Corduroy starts its processing by making a backup of the Java source code, and then creates a new Java file. It then begins parsing the original source code: any code without corresponding JML specifications, as well as plain (non-JML) comments, is written to the new file as-is. When JML specifications are detected before a method declaration, they are temporarily put aside until the method has been completely read by Corduroy.

To enable the metamorphic testing for a given function, Corduroy creates a new method that will execute all its tests. If the name of the original method was "foo", Corduroy creates a method "metaTestFoo" which returns boolean (to indicate whether it succeeded) and takes the same parameters as the original method "foo", so that they may be passed to "metaTestFoo" for the testing. Also, if the method "foo" returns anything other than void, the test method also takes a parameter called "result" which is the result of the execution of the original method; since "metaTestFoo" is called after "foo" has completed, the return value of "foo" can be used in the metamorphic testing without having to call the method again. The "metaTestFoo" method is declared protected so that subclasses may use it in their own testing.

Corduroy will then start to write the necessary code to the new Java source file. JML specifications and Java comments are written first; then lines containing the metamorphic properties are written as regular Java comments. Next, Corduroy adds a JML "ensures" clause (or an "also ensures" if the method overrides one in a superclass) that calls the test method and checks that it returns true, indicating success. There is only one call to the test method, regardless of the number of metamorphic properties, and the single test method checks all the metamorphic properties. By using an "ensures" clause to invoke the metamorphic tests, we convert the specification of metamorphic properties (using our extension to JML) to pure JML such that no modification to the JML implementation is necessary. At this point, Corduroy writes out the code for both the original method and new test method to the Java source file.

To allow for limited side effects in the tests, the new test code makes local copies of the variables that are listed as "assignable" (or "modifiable", which is a synonym) in the JML specification of the original method, and then restores them to their old values when the test method is done. For Objects, we check that they are Cloneable, so that the test code can use the "clone" method; if the class is not Cloneable, then it cannot be backed up. The test method is also declared as synchronized, so as to avoid any possible race conditions on these variables that may be caused by multiple threads accessing the same test method.

Although Corduroy allows for side effects with respect to variables labeled as "assignable", it does not actually ensure that the method is "pure", *i.e.* has no other side effects due to calling other methods. Obviously, it is possible that the runtime assertion checking may affect other parts of the application during its testing, so that other defects would be hidden or false positives would be revealed. To address this limitation, we are currently integrating Corduroy with Invite [13], a testing framework that allows for the execution of tests from within a running application, but in a separate sandbox so as not to affect its state. This will allow this approach to be used not only for testing in the development environment, but also for runtime testing in the deployment environment as well.

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

Figure 6. Example of metamorphic property expressed by extended JML specification.

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

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

Figure 7. Example of specification from Figure 6 after processing by Corduroy.

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

As an example, consider a function in a BankAccount class that calculates compound interest, but as a side effect also updates the BankAccount's balance. One of its metamorphic properties is that if the amount of time is doubled and the value is multiplied by the principal, the result will be equal to the square of the original amount of interest. Figure 6 demonstrates an example of the specification written using our extension to JML; Figure 7 shows the code after it has been processed by Corduroy.

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

7. Case Studies

To demonstrate the feasibility of our approach, we applied it to some open-source Java applications that fall into the category of "non-testable programs". In particular, we looked at WEKA [41] and RapidMiner [1], which both provide Java implementations for numerous machine learning and data mining algorithms, and are popular tools for the development of Java machine learning applications.

Our testing involved the Naive Bayes, Support Vector Machines, K-Nearest Neighbors, and C4.5 implementations in WEKA 3.5.8, and the Naive Bayes implementation in RapidMiner 4.1. For each of these five applications, we first determined its metamorphic properties, using the approach described in [31]; note that this step did not even require viewing the source code or having knowledge of implementation details. We then annotated the corresponding methods with specifications using our extension to JML, used Corduroy to pre-process the source code, and then compiled it using the JML 5.6 compiler. Last, we used some of the data sets from the UC-Irvine Machine Learning Repository [32] to perform our testing, using the JML 5.6 runtime environment to execute the code; no command line options were set for the machine learning applications, so all defaults were used. Our approach did not require the modification of any of the original application code, however some code needed to be added to facilitate our testing (see "Observations and Analysis" below).

7.1. Machine Learning Fundamentals

Before presenting our findings, we supply some background about the machine learning terminology and the algorithms we investigated. Readers who are familiar with machine learning may skip this section.

In supervised machine learning, data sets consist of a collection of *examples*, each of which has a number of *attribute* values and, in some cases, a *label*. The examples can be thought of as rows in a table, each of which represents one item from which to learn, and the attributes are the columns of the table. The label indicates how the example is categorized. These applications execute in two phases. The first phase (called the *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 *classification phase*), the model is applied to another, previously-unseen data set where the labels are unknown. In a classification algorithm such as the ones we have investigated, the system attempts to predict the label of each individual example.

The algorithms that we selected were chosen to represent different approaches to machine learning classification. Support Vector Machines (SVM) [38] and K-Nearest Neighbors (KNN) [3] are numerical approaches that treat each example in the training data as a point in N-dimensional space (where N is the number of attributes). When SVM generates a model, it creates a hyperplane that separates the points into classes, and then in the classification phase sees where each point in the data lies with respect to that hyperplane (*i.e.* which "side" of the hyperplane it is on). In KNN, the model is simply the coordinates of the points representing the training data, and an example is classified by looking at the K closest points (in N-dimensional space) and using a majority rules approach to decide on the classification.

Naive Bayes [26], by comparison, is a probabilistic approach that assumes independence between the attributes; the model is a formula that considers each attribute value and weighs it by its likelihood of correlating with the label. Lastly, C4.5 [34] is an approach that builds a decision tree, in which branches represent conjunctions of attribute values and leaves represent how the example is to be classified.

7.2. Findings

We specified a total of 25 metamorphic properties for the five applications we investigated, as described in Appendix A. This section describes our most interesting findings.

Our analysis of all the algorithms indicates that, in the training phase, they theoretically should produce the same model regardless of the input data order. That is, permuting the order of the examples should not affect the model, which only is concerned with the group as a whole. However, we did discover an inconsistency in WEKA's SVM implementation in which permuting the training data causes it to create different models for different input orders. For any non-trivial data set, in fact, this occurred even when all attributes and labels were distinct - thus removing the possibility that ties between equal values would be broken depending on the input order. An ML researcher familiar with this algorithm told us that because it is inefficient to run the quadratic optimization algorithm on the full data set all at once, most implementations perform "chunking" whereby the optimization algorithm runs on subsets of the data and then merges the results [37]. However, this is one important area, revealed by metamorphic testing, in which the implementation deviates from the expected behavior.

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

We 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 was a known defect in the version we tested, and was fixed in a later release.

Additionally, the two numeric algorithms, KNN and SVM, both have a metamorphic property as follows. Assume every attribute of the training data set *T* is numeric, and produces a model *M*, which when applied to a new example *e* produces a classification *c*. Now assume that each attribute of *T* is multiplied by -1, to produce model *M'*. If all attribues of *e* are also multiplied by -1, the classification *c'* is expected to be the same as *c*. The reason is that, since both KNN and SVM treat attribute values as coordinates in N-dimensional space, negating all of the values simply rotates the N-dimensional space around its origin point. In KNN, the nearest points are still the nearest, and in SVM, the hyperplane would also have been rotated, so the classification should stay the same.

Our testing did not demonstrate any defects in the KNN or C4.5 implementations in WEKA; although the tests cannot demonstrate correctness, either, since the correct output cannot be known in advance, the fact that the tests passed at least increases confidence in the implementations.

7.3. Observations and Analysis

The use of JML 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 both WEKA and RapidMiner, all of the classes we evaluated contained methods that took a single example as input and produced a classification as output. However, in the WEKA applications' 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 JML specification.

However, to work around this restriction, we found that it was rather straightforward to write a new test method that 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. This method could then be invoked via a JML "ensures" clause. Although the Corduroy framework was not used in these particular cases to generate the test code from the JML specification (the two WEKA defects were found using this approach, in fact), the overall testing approach of using metamorphic testing on a "non-testable program", enabled by JML runtime assertion checking, still proved to be useful.

A simpler workaround was to create a new method that would take the training data as a parameter, pass it to the method to generate the model, call the method to get that model, and then return the model. This very short method could then be used as part of the specification of the metamorphic properties of the method that builds the model.

In summary, with very little new code, and no modification of existing code, we were able to create complex metamorphic tests for the five applications and demonstrate some inconsistencies and defects in three of them. Although in this case we have only focused on machine learning classification algorithms, in [31] we demonstrated that the metamorphic testing approach would also work for supervised ranking algorithms and for unsupervised machine learning, such as intrusion detection systems.

7.4. Effect on Testing Time

Given that (depending on the size of the input) some machine learning applications can take hours or even days to run, and given that our approach calls for methods to be executed multiple times, the impact on testing time becomes an immediate concern. It would appear undesirable to have an approach which could conceivably multiply the overall processing time by doing everything over and over.

However, we note that by specifying the metamorphic properties of individual functions, rather than of the whole application, it is only those functions that are executed multiple times, so the overall impact may not be as great.

For instance, we conducted an experiment using the UC-Irvine Machine Learning Repository "Census Income" training data set, which contains 32,581 examples. To determine which phase of the algorithms complete faster, we did both training and classification using this data set for the four WEKA algorithms investigated (tests were done on a Linux Ubuntu 2.7.1 server with a dual-core 3GHz CPU and 1 GB of memory). Table 4 demonstrates that C4.5 and SVM spend much of their time in the training phase, whereas KNN and Naive Bayes spend more time in the classification phase, though all to different extents. This indicates that the overall impact on testing time will be reduced if the metamorphic testing is focused on the portions that take less time, and our own testing reflected similar behavior.

Algorithm	Total time	Training time	Classification time
KNN	276.91s	1.07s (0.3%)	275.84s (99.7%)
Naive Bayes	1.95s	0.59s (30.2%)	1.36s (69.8%)
C4.5	9.29s	8.74s (94.0%)	0.55s (6.0%)
SVM	4694.08s	4693.05s (99.9%)	1.03s (0.1%)

Table 4. Training and classification times for "Census Income" data set

Note that if JML runtime assertion checking is disabled, or if the application code is compiled using a regular compiler (*i.e.* not the JML compiler), then there is no testing performed as the application executes, so there is no performance overhead whatsoever.

8. Limitations and Future Work

To address the issues (described in Section 7) with existing code that is not written in a pure input/output fashion, we are considering modifying the extensions to JML so that a developer can specify a more complex sequence of metamorphic properties, instead of one simple boolean statement. However, given the workaround described above, we feel that the current approach itself is still sufficient to allow for metamorphic testing of such functions, albeit with some extra burden on the developers. Also, our approach currently only considers the metamorphic properties of individual methods. We are planning on extending the framework so that the metamorphic properties of an entire *application* can also be specified, perhaps using a more complex language than what is permissible in JML. However, this would make fault localization more difficult, which is a bit more straightforward in our current approach, given that we know which method's metamorphic property did not hold.

Of course, JML only applies to Java, but we are currently investigating the development of a Corduroy framework for C programs, perhaps based on a specification language like SpecC [19]. And due to our use of JML, our current approach requires access to the source code so that it can be recompiled with the specifications in place to enable runtime assertion checking. A system like [20] could conceivably be used to dynamically insert the Corduroy-generated metamorphic tests into already-compiled code.

Additional future work may also include the automatic detection of metamorphic relationships, similar to the work that has been done in discovering likely program invariants [17] and algebraic properties [24]. It could also be argued that static analysis of the code may be able to determine whether these properties hold, and we have begun preliminary investigations. Further research will be required to determine what are the limits for such approaches when detecting and checking these metamorphic properties.

9. Conclusion

In this paper we have presented an approach to testing software without test oracles that combines metamorphic testing, runtime assertion checking, and the JML specification language. We have also presented an implementation called Corduroy, and have demonstrated its feasibility in testing applications in the domain of machine learning; others have shown that metamorphic testing is suitable for other types of non-testable programs as well [10], and our work here improves on that by building upon a well-known, widely available specification language in order to specify the metamorphic properties, and automating the generation of test input data and invocation of the tests. This approach could also be used in the testing of applications for which there *is* a test oracle, and in particular would assist in creating new test cases (based on previous failed tests) for future regression testing and program evolution.

Addressing the testing of applications without oracles has been identified as a future challenge for the software testing community [4]. We hope that our findings here help others who are also concerned with the quality and dependability of such non-testable programs.

10. Acknowledgments

The authors would like to thank T.Y. Chen and Steve Lianoglou for their assistance. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

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

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

[3] D. Aha and D. Kibler. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

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

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

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

[7] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: architecture and early results. In *Proc of the 2007 conference on Specification and verification of component-based systems*, pages 47–53.

[8] 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.

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

[10] 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.

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

[12] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proc of the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328, June 2002.

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

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

[15] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254, 2006.

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

[17] 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.

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

[19] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC*. Springer, 2000.

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

[21] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[22] G. Haddad and G. T. Leavens. Extensible dynamic analysis for JML: A case study with loop annotations. Technical Report CS-TR-08-05, School of Electrical Engineering and Computer Science, University of Central Florida, 2008.

[23] 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.

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

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

[26] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proc of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995.

[27] S. W. Kim, M. J. Harrold, and Y. R. Kwon. MUGAMMA: Mutation analysis of deployed software to increase confidence and assist evolution. In *Proc of the Second Workshop on Mutation Analysis*, 2006.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

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

[36] S. Sankar, A. Goyal, and P. Sikchi. Software testing using algebraic specification based test oracles. Technical Report CSL-TR-93-566, Stanford Univ., 1993.

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

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

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

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

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

[42] G. Xu and Z. Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In *Proc of the 3rd IEEE ASE workshop on Formal Approaches to Testing of Software (FATES 03)*, 2003.

A. Metamorphic Properties Investigated

Here we list all of the metamorphic properties we investigated for each of the five applications tested in Section 7.

A.1. WEKA C4.5

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 multipled by a positive constant, and an example to be classified is also multipled by the same positive constant, the classification should be the same

4. If all attribute values in the training data are increased by a positive constant, and an example to be classified is also increased by the same positive constant, the classification should be the same

A.2. WEKA Naive Bayes

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. Updating a model with a new example should yield the same model created with training data originally containing that example

4. If all attribute values in the training data are multipled by a positive constant, the model should stay the same

5. If all attribute values in the training data are increased by a positive constant, the model should stay the same

A.3. WEKA KNN

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. Updating a model with a new example should yield the same model created with training data originally containing that example

4. If all attribute values in the training data are multipled by -1, and an example to be classified is also multipled by -1, the classification should be the same

5. If all attribute values in the training data are multipled by a positive constant, and an example to be classified is also multipled by the same positive constant, the classification should be the same

6. If all attribute values in the training data are increased by a positive constant, and an example to be classified is also increased by the same positive constant, the classification should be the same

A.4. WEKA SVM

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 multipled by -1, and an example to be classified is also multipled by -1, the classification should be the same

4. If all attribute values in the training data are multipled by a positive constant, and an example to be classified is also multipled by the same positive constant, the classification should be the same

5. If all attribute values in the training data are increased by a positive constant, and an example to be classified is also increased by the same positive constant, the classification should be the same

A.5. RapidMiner Naive Bayes

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 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

4. If all attribute values in the training data are multipled by a positive constant, the model should stay the same

5. If all attribute values in the training data are increased by a positive constant, the model should stay the same