

# A Specification Coverage Metric Using a Dynamic Invariant Detector

Rob McGregor  
Dept. of Computer Science  
The College of William & Mary  
Williamsburg, VA 23185  
rlmcgr@cs.wm.edu

Will Thomasson  
Dept. of Computer Science  
The College of William & Mary  
Williamsburg, VA 23185  
wethom@cs.wm.edu

**Abstract**—Traditional coverage metrics measure the ability of test cases to fully exercise the source code. The assumption is that fully covered code will better test the correct behavior of the software. Unfortunately, there is no guarantee that the software will meet its specification. In this paper, we present a new coverage metric which measures the ability of the test cases to fully exercise the *specification* of the software. Our approach allows the developer to iteratively develop and elaborate the test suite and the specification together. We infer a specification from the execution of the software for the given test cases. If the test cases are inadequate, the specification will be inaccurate, and if the code is incorrect, the inferred specification will be incorrect. The tester can then refine the test suite or repair the software to improve the resulting specification. To evaluate this approach we applied it to three Java classes and their associated test suites. For each of the three classes, our method revealed incorrect and missing invariants in our specification, which implied limitations in our test suite. We were subsequently able to improve the test suite (and specification) by adding appropriate test cases. After applying our method several times, all incorrect invariants were removed, and our final test suite was more complete than the original.

## I. INTRODUCTION

Developing an adequate set of test cases for computer programs is difficult. If a given set of tests on a software system fails to test a part of the code that contains an error, then the error may occur sometime while the software is being used in practice, which can cause severe consequences. While exhaustively testing a program can show it to work under all conditions, doing so is often impractical, if not impossible. Programmers are often willing to instead accept that there might be a few bugs rather than spend all of their time writing test cases. It is therefore beneficial to come up with a technique that can be used to improve test suites by locating potential shortcomings of existing suites, thereby making clear what types of test cases need to be added.

Most test coverage metrics are based on the number of branches of the code that have been tested, or in other words, how much of the code has been tested. However, we note that fully covered code must not just test all branches of the code, but should test the correct behavior of the software to make sure it meets its specification. Therefore, we have developed a coverage metric for a test suite based on the ability of the test cases to fully exercise the specification of the software. If

an inferred specification from the execution of the software for given test cases is inaccurate, it will hint at either an incomplete test suite or incorrect code.

This research allows the specification to be developed during the process of improving the test suite, requiring no predefined specification from the developer. This is accomplished by using dynamic program invariant detection to infer the invariants of the program. These invariants form a specification of the program which can then be analyzed for correctness. Because the analysis is done dynamically, invariants are generated based only on those test cases that are given to the program invariant tool, and therefore their accuracy relies on the ability of the test suite to accurately portray the specification of the code. A lack of code coverage by the test cases will result in inaccurate invariants being produced, and these inaccuracies will suggest an insufficient test suite.

Our method works iteratively, so that at each step a new specification is produced by inferring the program invariants. Analyzing the invariants in the inferred specification can suggest cases that should be added to the test suite in order to improve the accuracy of the invariants found on the next iteration. As the accuracy of the invariants continues to improve, an implied improvement in the test suite will result, providing a relatively easy method of improving software testing.

By developing a well-defined methodology for improving test suites using invariant detection, we provide a concrete process through which software engineers can develop superior test suites for their software. These test suites would exhibit efficiency: they would include an adequate number of test cases to cover the specification, but not so many test cases that exhaustively testing them would be infeasible. We show that this method is in fact feasible by successfully applying it to three test suites for Java classes.

We created three Java classes to which we applied our method. For each of the three classes, we collected data on the number of incorrect and missing invariants on each iteration, as well as on the number of test cases we added each iteration. We found that we were able to eliminate all the incorrect invariants within four iterations, leading to a correct specification being output by the program invariant detector

on the final run for each of the classes.

The rest of this paper is organized in the following fashion: section II gives background information on invariants, test suites, code coverage techniques, and the Daikon tool. Section III gives an overview of our approach. Sections IV through VI describe the classes to which we applied our method, and our experiences while applying it to each. Section VII presents the results of our work, including tables listing the number of incorrect invariants found and tests added on each run for each class. The evaluation of our method can be found in section VIII. Finally the conclusion follows in section IX.

## II. BACKGROUND

In this section, we describe the background ideas and theories used in this paper. This knowledge is useful for understanding the purpose behind our research.

### A. Program Invariants

A key idea in program analysis is that of program invariants. A program invariant is a rule that must always be true under any possible execution of the program within the domain for which it is defined. For example, an invariant for the exit of a function must be a statement or relationship among program variables that is true at the exit point of the function for all executions of that function. Because program invariants describe a program in its execution, they can be used as a type of program specification.

Program invariants traditionally are detected by manually following the trace of a program, searching for patterns between the variables. For example, if  $x=1$  and  $y=1$  before a loop, and inside the loop are the statements  $x=x+1$ ; and  $y=2*y$ ; then the invariant over the loop is  $y \geq x$ . Another example is a linked list: for a node in the middle of the list, if the nodes after it in the list are never deleted, its pointer to the next node is never null.

Program invariants have many applications. One use includes using the invariants to trim unnecessary code; for example, if an invariant coincides with the test condition of an `if` statement, the `if` will always evaluate true, so it can be eliminated. Another use is to implement the invariants as `assert` statements so that if changes are made to the code, the changes will not violate the invariants. Program invariants can also be inserted into code as comments to improve understanding of the code; this improved understanding is very beneficial for maintenance of legacy code. A further useful application of program invariants is evaluating test cases; for example, if an invariant states that  $x < 100$ , and all of the test values for  $x$  are less than 100, the test suite will not help in finding bugs with code involving the variable  $x$ . Upon realizing the inadequacy of the test suite, the programmer can add test cases, but more importantly, the programmer now knows exactly what sort of test cases to add. In the example above, the programmer needs to add a test case in which the value of  $x$  is greater than 100 [1].

### B. Test Suites

Another important idea in program analysis is the concept of test suites. Because programs often have very large numbers of inputs, exhaustively testing each input is infeasible. To test the program in a manner that covers all inputs, but is also efficient, programmers use test suites. The common notion of a test suite is a subset of all possible input, well-chosen to demonstrate all the possible behaviors of all inputs. For example, if the `main` function in the program included the following code: `if(x mod 2 == 0) print x;`, then the test suite need not contain all possible values of  $x$ ; it should contain two different values for  $x$ : one even, and one odd.

The test suite must be large enough that it represents all inputs during testing, but small enough that it can be exhaustively tested. Unfortunately, choosing a test suite proves a challenge for all but the most trivial program.

### C. Code Coverage Techniques

Because of the importance of testing software and ensuring that the software is as adequately tested as possible, there are many techniques that attempt to create test cases that will test a program as thoroughly as possible. Two common methods of creating test data are “black box”, also known as functional analysis and “white box”, which is also known as structural analysis [2]. Black box testing uses the external specification of the software, viewing the internal organization as a black box, meaning the method does not care about how the program works, but rather what it does, in deciding the tests to create. White box testing, on the other hand, relies on the structure of the code itself in determining the best test cases to create.

Since exhaustive testing is practically impossible, black box methods use techniques such as boundary value analysis [2], which tests the code at the boundaries of allowable inputs, where errors are most likely to be located. Another functional technique is called design-based testing, which creates a tree structure representing all functional capabilities of the program. The goal is to have the tree branch into all possible areas, but this can be quite difficult to accomplish [2]. A final functional analysis method to create good test cases is called Cause-Effect Graphing [2]. This technique uses the high-level specifications to partition the possible outputs into classes based on the type of input, and then tries to test each possible class of output.

White box testing usually attempts to find tests that will cover as many of the branches of the code as possible. This can be accomplished by representing the program by a graph with the vertices representing statements and the edges representing control flow. Tests are then created that attempt to cover all possible paths. Another structural testing technique is known as complexity-based testing [2]. This is a complicated testing strategy that first uses a complexity metric called cyclomatic complexity to measure the complexity of the program. Then, the actual complexity is determined to be the complexity of the program while running on the test suite. The goal is to grow the test suite so that the complexity of the program while

running on it becomes as close to the cyclomatic complexity as possible.

#### D. Daikon Tool

The dynamic program invariant detector used in our experiments was the Daikon invariant detector [1], [3]. Daikon runs a program with its test data as inputs to attempt to find patterns and relationships among the computed values. Daikon uses dynamic techniques for invariant detection because sufficiently detecting invariants cannot be done with static techniques alone. Pointers and other dynamically-allocated objects, for example, cannot be traced statically. Like many dynamic techniques, Daikon first analyzes the code statically, instrumenting the code with valuable output information at key points. It then runs the instrumented version over the test cases to infer the invariants of the program.

Because Daikon relies on the test cases provided to it, its detection of invariants is only as effective as the test cases [3]. If test cases are limited in their scope of covering all possible inputs, the invariants will be limited in explaining the execution of the program. Similarly, if the set of test cases fully exercises the specification of the software, the invariants will have perfect accuracy in describing the execution of the program (of course, the latter is an ideal case that cannot realistically be achieved, but it is still a goal that can nearly be attained). This dependence on the test cases defines a relation that can be used to examine the invariants given the test cases, or to examine the test cases given the invariants; we will perform the latter in our research.

### III. APPROACH

In this section, we present a brief overview of our method.

#### A. Develop an Initial Test Suite

Before the dynamic invariant detector could be run on our code, we needed to provide it with test cases. For each class, we created a separate tester file that tested each function with random input, usually ten tests for each function. Some of the functions, such as constructors, were not tested explicitly, but were indirectly called by other functions.

Before running the dynamic invariant detector, we created our own list of intuitive invariants for each function. We wanted to have a good idea of the specification before we saw the output from the invariant detector, and this list also helped us detect missing invariants.

#### B. Establish and Analyze the Specification

Once the initial test suite was created, we ran the dynamic invariant detector to derive the specification. The invariants returned from the invariant detector would indicate whether or not there were flaws in the specification or in the coverage of the test cases.

After deriving the specification from the dynamic invariant detector, we assessed the quality of the specification by analyzing the invariants; if incorrect invariants were found, there were inadequacies in the test suite, in the specification,

or in both. Because an invariant is a universal truth, a single counterexample suffices to disprove it. For each incorrect invariant that was found, we listed its counterexample beside it in our documentation.

For each method in the class, we created three lists: one of correct invariants, one of missing invariants, and one of incorrect invariants. To fill the lists, we examined each invariant that the invariant detector produced; if we knew the invariant to be true, we placed it in the list of correct invariants. If we knew that it had a counterexample, we placed it in the list of incorrect invariants and wrote its counterexample as an annotation beside it. If at first we were not sure whether the invariant was correct, we closely examined the code of the method for which the detector had generated the invariant until we were able to discern its correctness.

We then compared our list of invariants that we created prior to the execution of the dynamic invariant detector to the invariants output by the detector; any invariants in our list that were not included in the output were added to the list of missing invariants.

#### C. Add Test Cases

If incorrect invariants were found, we added test cases to each function for which incorrect invariants had been discovered. These test cases were chosen specifically to satisfy the counterexamples we listed in the previous step, and the test cases were chosen to minimize the number of test cases added; if a certain test case was able to satisfy several counterexamples at once, it was chosen to be added.

#### D. Cycle and Conclude

The previous three steps are repeated until no further incorrect invariants are found. Missing invariants may remain, because they may indicate more about the limitations of the dynamic invariant detector than about the limitations of the test suite.

### IV. THE EXPERIMENT: DATE

#### A. Overview of the Date Class

The second class to which we applied our method was a `Date` class, originally created by Mark Allen Weiss [4], and modified and expanded by the authors of this paper. It has private integer data members `month`, `day`, and `year`. There is also a global (zero-indexed) array `int monthDays[]` of size 12 that contains the number of days in each month (i.e. `monthDays[11] == 31` signifies that December has 31 days). The minimum date allowed is January 1, 0000; this date is considered to have a magnitude of one day.

The functions in the `Date` class include:

`Date(int theMonth, int theDay, int theYear)` - a constructor that initializes a new date

`boolean equals(Object rhs)` - returns true if the invoking date and `rhs` are the same

`int Subtract(Date rhs)` - returns the number of days between the invoking date and `rhs`; if the number of days is negative, an error message is printed

`Date Subtract(int days)` - returns the date that is `days` prior to the invoking date; if `days` is greater than the magnitude of the invoking date, an error message is printed

`Date Add(int days)` - returns the date that is `days` after the invoking date; if `days` is negative, an error message is printed

`Date daysToDate(int numDays)` - returns the date whose "magnitude" is `numDays`; for example, December 11, 0001 has a magnitude of  $366 + 345 = 711$  days

`int dateToDays(Date d)` - returns the magnitude of `Date d`

`Date midAnniversary(Date input)` - returns the date six months from `Date input`

`boolean isLeapYear(int year)` - returns true if `year` is a multiple of 4

### B. Develop an Initial Test Suite

We also created a `runRandom()` function for the `Date` class. In it, we created two random dates and used them to test each function one to four times. Then, `runRandom()` was called 10 times. These random test cases provided the basis for our test suite.

As the reader may recall, we created a list of intuitive invariants for each function before running the invariant detector, to aid us in identifying missing invariants. An example of one of the invariants in that list is an exit invariant for the function `Add(int days)`, which states that the year of the returned date is greater than or equal to the year of the invoking date. In Daikon notation, this would be written as `return.year >= this.year`.

### C. Establish and Analyze the Specification

After creating the initial test suite for our class, we ran the invariant detector on the test suite and the source code for the class to establish the specification. We then analyzed the invariants to determine the accuracy of the specification.

When Daikon was applied to the `Date` class function `dateToDays(Date d)`, for example, several invariants were easily identified as correct. These included:

```
d.day <= d.monthDays[d.month-1]    on entry
d.month >= 1                        on exit
```

As the reader may recall, `monthDays[]` stores the number of days in each month, and is zero-indexed. Because `month` ranges from 1 to 12, it is necessary to access the `monthDays[]` array with `month-1`. So, the first invariant tells us that, for example, the number of days in month 12 must be less than or equal to 31.

Several other invariants were immediately recognized as incorrect. These included the exit invariants:

```
d.year > d.monthDays[d.month-1]    (Invariant Set 1)
return > d.monthDays[d.month-1]
```

The first is clearly incorrect because the input year can be less than 28 (the smallest entry in `monthDays`). The second is also incorrect, because a date in January, year 0 would return a value less than or equal to 31.

Examples of less obvious invariants include the following exit invariants:

```
d.year < return    (Invariant Set 2)
d.month < return
d.day < return
```

After examining the code for `dateToDay(Date)`, we were able to determine the correctness of each of these invariants. We found that the minimum value of `return` is one because the earliest possible date is January 1, 0000, which is defined to have a magnitude of one. Given that `year` has a minimum value of zero, and that it grows much more slowly than the `return` value, `d.year` will always be less than `return`, so we realized that the first invariant is correct. We also realized that the second and third invariants are incorrect: January 1, 0000 provided a counterexample because both `d.month` and `d.day` have values of one, which is equal to the `return` value when that date is input. The first invariant was added to the list of correct invariants, while the second and third were added to the list of incorrect invariants with their counterexamples listed beside them as annotations.

Some of our predetermined invariants were missing from Daikon's output. Examples of these include the following:

```
d.month <= 12    on entry    (Invariant Set 3)
d.day >= 1       on entry
d.year >= 0      on entry
return >= 1     on exit
```

These were added to the list of missing invariants.

### D. Add Test Cases

In the previous section we mentioned that after the invariants found by Daikon were analyzed, we created a list of the incorrect invariants and added their counterexamples beside them as annotations. At this step, we created a list of inputs to each function that would satisfy those counterexamples. Because invariants are universal statements, a counterexample is all that is necessary to show them to be false. When creating the inputs, we tried to choose inputs that would satisfy more than one counterexample at a time in order to minimize the number of test cases necessary to improve the coverage our code. However, it is also necessary to make sure that the invariant is fully shown to be false, as illustrated in the next section, section E: Cycle and Conclude.

An example of our handling of incorrect invariants is in the function `dateToDays`:

```
d.year > d.monthDays[d.month-1]    (Invariant Set 4)
return > d.monthDays[d.month-1]
```

the counterexamples are:

```
d.year <= d.monthDays[d.month-1]
```

```
return <= d.monthDays[d.month-1]
```

An input of any day in January in the year 0 would satisfy the counterexamples to Invariant Set 4 and demonstrate both of the invariants to be false, so we added January 1, 0000 to the test cases for `dateToDays`. This same input invalidates the less obviously incorrect bottom two invariants from Invariant Set 2 (these invariants should have `<=` instead of `<`).

Adding test cases with the intent of eliminating the incorrect invariants often aids Daikon in finding the missing invariants as well because Daikon keeps track of the number of instances in which each possible invariant holds. A higher number of instances for a specific condition increases Daikon's confidence that the condition will always hold true, and therefore will be output as an invariant.

For example, the `Date` class function `dateToDays(Date)` did not produce the invariants in Invariant Set 3 on the first run of the invariant detector, but subsequent runs of Daikon produced these invariants, even though we did not add any specific test cases to test for them. The reason that we did not add specific test cases is that we knew these held true for all test cases in our test suite, so adding test cases to eliminate the incorrect invariants would reinforce to Daikon the correctness of the invariants in Invariant Set 3. For example, the test case January 1, 0000 added above to eliminate incorrect invariants, as well as the other tests we added, will increase the number of instances of each condition stored by Daikon, improving Daikon's confidence in these missing invariants. Only in cases where we found that an invariant was missing because it had been untested did we add specific cases to test that invariant. The reason for being conservative is that there is no improvement gained in the quality of a test suite by simply adding test cases for things that have already been tested. By our last run, all of these missing invariants except for `d.month <= 12` were found by Daikon. The fact that this invariant was not found is simply due to a limitation in Daikon, rather than an incomplete test suite. Daikon tends to test for comparisons to 0 or 1 rather than any arbitrary integer such as 12.

It is not always as simple as adding one test case to eliminate all the incorrect invariants in a function. For most of our functions, five to ten new test cases were necessary.

### E. Cycle and Conclude

After the new test cases were added, we ran Daikon again on our new test suite. At this point, we followed the same steps as before, determining the correctness of the invariants and adding necessary test cases. It was not uncommon for new or changed invariants to appear as a result of our new test suite. For example, to remove the incorrect invariants in Invariant Set 4, we added the input January 1, 0000 because that date satisfies the counterexamples listed below Invariant Set 4. When we ran Daikon a second time though, we found that the incorrect invariants were not completely eliminated. Instead, Daikon had changed them to the following:

```
d.year != d.monthDays[d.month-1]
```

```
return != d.monthDays[d.month-1]
```

We quickly realized after seeing these invariants that we tested only for `d.year < d.monthDays[d.month-1]` and `return < d.monthDays[d.month-1]`; we now had to add test cases where `d.year == d.monthDays[d.month-1]` and `return == d.monthDays[d.month-1]`. As mentioned in the above section, section D: Add Test Cases, in determining which test cases must be added to disprove an incorrect invariant, it is necessary to fully disprove it, which we failed to do at first. If an invariant is not fully shown to be incorrect, then the number of runs necessary to eliminate all of the incorrect invariants will increase. On our second addition of test cases, we added January 1, 31 and January 31, 0000 to satisfy the above counterexamples.

Our example describes only a few invariants and one specific function. It is important to realize that we added the test cases for all the functions before running Daikon again, and for the sake of simplicity, we have not included in our example all the incorrect or missing invariants of this function.

## V. THE EXPERIMENT: MATRIX

### A. Overview of the Matrix Class

The first class to which we applied our method was a matrix class, created by the authors of this paper. It has two private data members: `int size` and `int [][] mat`, where `mat` is a `size x size` square matrix.

The functions in the `Matrix` class include:

`matrix(int capacity)` - a constructor that initializes a new matrix with `size` equal to `capacity` whose entries are all zero.

`boolean isEmpty()` - returns true if all of the entries in the matrix are zero; false otherwise

`void clear()` - sets all the entries in the matrix to zero

`int size()` - a public function that returns the private `size` of the matrix

`void set(int i, int j, int x)` - sets the element at `mat[i][j]` to `x`

`void setRow(int i, int x)` - sets all elements in row `i` to `x`

`void setCol(int j, int x)` - sets all elements in column `j` to `x`

`int element(int i, int j)` - returns the value of `mat[i][j]`

`void fillWith(int x)` - sets all elements in the matrix to `x`

`matrix negate()` - returns a matrix whose entries are the negation of the corresponding entries in the invoking matrix

`matrix SubMatrix(int x, int y)` - returns a copy of the invoking matrix with row `x` and column `y` removed

`matrix Add(matrix A)`, returns the result of adding the matrix `A` to the invoking matrix

`matrix Subtract(matrix B)`, returns the result of subtracting the matrix B from the invoking matrix by adding the negation of the matrix B to the invoking matrix

`matrix Multiply(matrix A)`, returns the result of multiplying the invoking matrix by the matrix A

### B. Develop an Initial Test Suite

For the initial test suite of the `Matrix` class, we created a function `runRandom(int k)` that creates a  $k \times k$  matrix and tests each `Matrix` class function four times: twice with random integer values and twice with values of zero. While it may appear redundant, The testing with values of zero was performed twice for each function to give our dynamic invariant detector additional data from which to more accurately infer its invariants. We called the `runRandom(int k)` function with integer values zero through eight for  $k$ , calling it three times for each integer, to test different matrix sizes. Originally, we tried testing the `Matrix` class with matrix sizes up to  $10 \times 10$ , but the test program runtime was too large, and although it completed after a while, the resulting dataset was too large for our invariant detector to handle.

### C. Establish and Analyze the Specification

After developing the initial test suite for the `Matrix` class, we established a specification by running the dynamic invariant detector. Our analysis of the specification for the `Matrix` class was simple compared to the other two classes: most invariants were correct on the first iteration. The ones that were incorrect were obvious, as in the following set of invariants for the function `set(int i, int j, int x)`:

```
i != x    (Invariant Set 5)
j != x
x != 0
```

We determined quickly that these invariants were incorrect, because  $x$  can be any integer. These were added to our list of incorrect invariants with their counterexamples. This process was done for all functions in the `Matrix` class.

### D. Add Test Cases

To eliminate the incorrect invariants in Invariant Set 5, thereby improving the specification, required only one test case: `set(0, 0, 0)`. Test cases for other functions were similarly added.

### E. Cycle and Conclude

Because of the test case above, the incorrect invariants in Invariant Set 5 were eliminated after the second run of Daikon. After the second run, no incorrect invariants remained, and the specification was determined to be correct.

## VI. THE EXPERIMENT: POLYNOMIAL

### A. Overview of the Polynomial Class

The final class to which we applied our method was a `Polynomial` class, written by the authors of this paper. The private data members are `int highPower` and `int coeffArray[]`,

which has size 100. `coeffArray` is an array that holds the coefficients of the polynomial; the indices of the array correspond to the powers of the polynomial. `highPower` indicates the index of the highest-indexed non-zero coefficient. For example, the polynomial  $5x^4 + x^2 + 2x + 7$  would have a coefficient array `{7, 2, 1, 0, 5}` (with the rest filled with 95 zeroes because its size is 100) and `highPower` equal to 4.

The functions in the `Polynomial` class include:

`Polynomial(int [] a)` - a constructor that initializes a new polynomial whose coefficient array is the reverse of the array `a`; this reversing is done so that creating a polynomial is more intuitive

`Polynomial(int c)` - a constructor that initializes a new polynomial whose coefficient array contains only the constant `c` at index 0

`void zeroPolynomial()` - sets all the elements in the coefficient array of the invoking polynomial to zero and also sets the `highPower` to zero

`Polynomial add(Polynomial rhs)` - returns the polynomial that is the sum of the invoking polynomial and `rhs`

`Polynomial multiply(Polynomial rhs)`  
- returns the polynomial that is the product of the invoking polynomial and `rhs`

`int evalAt(int c)` - returns the result of evaluating the invoking polynomial at the value `c`

`Polynomial derivative()` - returns the first derivative of the invoking polynomial

`Polynomial raisedTo(int k)` - returns the invoking polynomial raised to the  $k^{\text{th}}$  power

### B. Develop an Initial Test Suite

Just as we did for the `Matrix` and `Date` class, we created a `runRandom()` function for the `Polynomial` class. Inside this function, we created six random polynomials, with some of the coefficients randomly chosen to be zero so the polynomials would not always be “full”; that is, so that not all entries in the coefficient array would be nonzero. Each function in the `Polynomial` class was then tested four to twelve times within the `runRandom()` function: two to six times for the random polynomials, and then two to six more times for zero polynomials. The `runRandom()` function was called 10 times from the main function.

### C. Establish and Analyze the Specification

`Polynomial` was the most difficult class of the three to analyze. As Figure 4 shows, Daikon found a large number of invariants; these complex invariants were not always easy to read or to comprehend. For example, some of these compared the coefficient array of the invoking polynomial to the coefficient array of the polynomial parameter using `highPower` of the return polynomial as an index.

Some specific instances of the difficulty of the invariants from `Polynomial` include:

This exit invariant from `Add` (Invariant 1):

```
return.coeffArray[this.highPower] >=
this.coeffArray[return.highPower]
```

This exit invariant from `Multiply` (Invariant 2):

```
rhs.coeffArray[this.highPower] in
rhs.coeffArray[return.highPower..]
```

And this exit invariant from `Derivative` (Invariant 3):

```
return.coeffArray[return.highPower] ==
this.highPower * this.coeffArray[this.highPower]
```

For Invariant 1, Daikon examined the relationships between data members of `this` and `return`, causing our verification to be difficult. After some analysis, we realized this invariant is incorrect. For example, consider the case where `this` =  $2x^2 + 3x$  and `rhs` =  $-2x^2$ . With this input, `return` =  $3x$ , so we have `return.highPower` = 1, and `this.highPower` = 2. So, `return.coeffArray[this.highPower]` = 0 and `this.coeffArray[return.highPower]` = 3. Since  $0 < 3$ , this invariant is incorrect.

With similar analysis, it can be shown that Invariant 2 is incorrect, and that Invariant 3 is correct.

#### D. Add Test Cases

To eliminate the incorrect invariant Invariant 1, we added the test case mentioned above, where `this` =  $2x^2 + 3x$  and `rhs` =  $-2x^2$ . Eliminating Invariant 2 can be accomplished by adding the test case `this` =  $x$  and `rhs` =  $x$ . Test cases for other functions were similarly added.

#### E. Cycle and Conclude

We ran Daikon on the `Polynomial` class four times, and after the fourth run, all incorrect invariants were eliminated, resulting in an accurate specification.

## VII. RESULTS

In experimenting with our method of improving test suites, we used the method on increasingly complex classes. We started with the `Matrix` class, whose total number of invariants produced by Daikon as well as the amount of incorrect invariants was quite small. Additionally, the ability to determine the correctness of the output invariants was easiest for this class. For each of the other two classes, `Date` and `Polynomial`, more invariants were produced per function, and the complexity of these invariants was greater, making their analysis more difficult.

In general, the number of incorrect invariants produced in our first run for a given function was roughly proportional to the total number of invariants produced for that function. The functions that produced all the correct invariants on the first run usually had only four or five total invariants, often due to the fact that they were relatively simple or involved simple

operations. Another common experience encountered in most of the functions and classes was that we could consistently reduce the amount of incorrect invariants at each run by a large amount. In fact, for each of the classes, we had eliminated all of the incorrect invariants detected by Daikon within four iterations of our method. The convergence of the number of incorrect invariants to zero increased as we became more familiar with the types of invariants produced by Daikon. For example, earlier in our testing, we would have attempted to disprove an incorrect invariant of the form  $A < B$  with a counterexample showing that  $A > B$  is possible, omitting the case where  $A == B$ . However, later in our experimentation, to avoid Daikon producing a new invariant of the form  $A! = B$ , we added an additional test case showing that  $A == B$  was possible at the same time we added the  $A > B$  test case, which decreased the number of invariants produced at the next step.

For each of the three classes, we recorded the number of incorrect invariants Daikon detected, and the number of known invariants that were missing from Daikon's output. Figures 1, 2, and 3 list the number of invariants found on each run of our method.

Before each run (not including the first), we added test cases to eliminate the incorrect invariants found on the previous run. The number of incorrect invariants on each run decreases from run to run because, although some new ones were detected on each run, more were eliminated.

After the last run shown, all incorrect invariants were eliminated. The number of missing invariants was usually decreasing as well, except for certain cases where it stayed the same. In these cases, even though we added input to aid Daikon in detecting the missing invariants, it was unable to detect them. Most of these were invariants that were undetectable by Daikon because they were either too complex or dealt with items whose relationships Daikon does not examine.

An example of an invariant we determined, but was too complex for Daikon to detect, was the exit invariant `(return == true) ==> ((year % 4) == 0)` in the `Date` class function `isLeapYear(year)`. Daikon does not test for relationships using the `%` (*mod*) function. Another invariant that Daikon missed that would have been impossible for it to detect is the exit invariant `return.mat[i][j] == x` in the `Matrix` class function `set(i, j, x)`; Daikon misses this invariant because it examines specific entries only in one dimensional arrays, and does not have the functionality to examine both dimensions in a two-dimensional array simultaneously.

#### A. Matrix Class

As is evident in Figure 1, Daikon found few incorrect invariants for the `Matrix` class. This lack of invariants was generally the result of Daikon's inability to properly analyze two-dimensional arrays. Most of the invariants that were discovered by Daikon for this class were simple, obvious invariants because the relationships involving the two-dimensional arrays were too complex for it to analyze. All incorrect invariants were eliminated after the first run by adding three specifically

Function Name	First Run	
	Incorrect	Missing
<code>matrix()</code>	0	0
<code>matrix(int)</code>	0	0
<code>isEmpty()</code>	0	0
<code>clear()</code>	0	0
<code>size()</code>	0	2
<code>set(int, int, int)</code>	7	1
<code>setRow(int, int)</code>	6	1
<code>setCol(int, int)</code>	6	1
<code>element(int, int)</code>	5	2
<code>fillWith(int)</code>	0	1
<code>negate()</code>	0	0
<code>SubMatrix(int, int)</code>	0	0
<code>Add(matrix)</code>	0	1
<code>Subtract(matrix)</code>	0	1
<code>Multiply(matrix)</code>	0	0

Fig. 1. Incorrect and Missing Invariants for the `Matrix` Class

chosen test cases to each of the functions with incorrect invariants.

### B. Date Class

Looking at Figure 2, one may notice that it is odd that `equals(Date)` had an increasing number of incorrect invariants between the first and second runs. It turns out that this increase was the result of our not adding sufficient test cases, as we see below.

The incorrect invariants found on the first run were the following:

```

    this.month < this.year
    this.year > monthDays[this.month-1]
    this.day < this.year
    this.year > monthDays[this.month-1]
    return == false

```

For each invariant above of the form  $A < B$ , we added a test case that would show that it is possible for  $A > B$ ; likewise, for invariants whose form was  $A > B$ , we added test cases that would demonstrate that  $A < B$  is possible. We were surprised to find that, on the second run, the invariants were still in place, but all of the ones of the form  $A < B$  and  $A > B$  had changed to  $A \neq B$  because we had not provided test cases that would demonstrate that  $A == B$  is possible. Because we were alerted to this situation while examining this function, we took precaution in the future so that whenever we encountered an incorrect invariant of the form  $A < B$ , we added test cases that showed that both  $A > B$  and  $A == B$  were possible.

There is one incorrect invariant in the list above that was not of the form  $A < B$  or  $A > B$ : `return == false`. After seeing this invariant, we realized that we had not tested `equals(Date)` with a situation where the invoking date was equal to the date passed to the function. This was due to our testing method of running each function ten times with random input. Because an invariant is a universal statement, a counterexample suffices to disprove it. So, we added a

single test case in which the invoking date and the parameter date were the same. The output from Daikon included three incorrect invariants that had not appeared on the previous run:

```

    (return == true) <==> (this.year == 15)
    (this.year == 15) ==> (this.day == 31)
    (this.year == 15) ==> (this.month == 1)

```

The date we chose for our test case was January 31, 0015, leading Daikon to assume this was the only date that could result in a return value of true. To remove these incorrect invariants, we realized we would need more than one test case in which the invoking date is equal to the date passed to the function. We added four specific and 20 random test cases for this situation, and after running Daikon, the three incorrect invariants listed above were removed. Four incorrect invariants remained, but they were unrelated and were removed on the fourth run.

It is important to note that the situation we encountered with `equals(Date)` is not the typical situation in our method; usually, our method results in a decrease in incorrect invariants between each run. For example, in Figure 2, the data for the `Date` class function `Subtract(int)` demonstrates a steady decrease in incorrect invariants between iterations, as well as the elimination of a missing invariant.

There was no decrease in the number of missing invariants after the second run because although we added test cases that would demonstrate situations in which the missing invariants were true, these were types of invariants that Daikon does not detect.

### C. Polynomial Class

Because `Polynomial` was the last class to which we applied our method, we were aware of pitfalls we could encounter while applying our method (such as the one mentioned in the `Date` class function `equals(Date)`), and took precautions to avoid them. Using these precautions, and despite the greater complexity of the `Polynomial` class, we were able to achieve

Function Name	First Run			Second Run			Third Run		
	Incorrect	Missing	Tests Added	Incorrect	Missing	Tests Added	Incorrect	Missing	Tests Added
Add(Date)	31	5	3	25	5	3	0	4	0
Date(int, int, int)	3	4	N/A	1	3	N/A	0	3	N/A
Subtract(Date)	47	6	3	36	6	8	0	6	0
Subtract(int)	28	7	4	13	6	3	0	6	0
dateToDays(Date)	7	4	1	5	2	2	0	1	0
daysToDate(int)	6	5	2	1	5	1	0	3	0
equals(Date)	5	13	1	7	13	4	4	12	4
getYear()	6	4	1	6	4	2	0	4	0
isLeapYear(int)	0	4	0	0	4	0	0	4	0
midAnniversary(Date)	17	0	2	12	0	4	1	0	1
toString()	4	0	1	4	0	2	0	0	0

Fig. 2. Incorrect and Missing Invariants for the Date Class

Function Name	First Run			Second Run			Third Run		
	Incorrect	Missing	Tests Added	Incorrect	Missing	Tests Added	Incorrect	Missing	Tests Added
Polynomial()	0	0	0	0	0	0	0	0	0
Polynomial(int [])	7	0	2	0	0	0	0	0	0
Polynomial(int)	4	0	0	0	0	0	0	0	0
zeroPolynomial()	2	0	0	1	0	0	0	0	0
add(Polynomial)	38	0	8	6	0	3	0	0	0
multiply(Polynomial)	39	1	8	4	1	4	0	1	0
evalAt(int)	1	0	1	0	0	0	0	0	0
derivative()	3	0	4	0	0	0	0	0	0
raisedTo(int)	24	0	12	12	0	16	2	0	2

Fig. 3. Incorrect and Missing Invariants for the Polynomial Class

a better convergence to zero for the number of incorrect polynomials between each run. Polynomial required only three runs to eliminate the incorrect invariants from all but one of its classes: the raisedTo(int) function. This is the most complex of the Polynomial functions (it uses three return statements), and required one additional run to eliminate all the incorrect invariants.

## VIII. EVALUATION

Our experiment did largely what we expected of it. For each of the four classes with which we experimented, the original specification derived from the detection of program invariants showed that our tests did not at first fully cover the specification. We were able to use the incorrect invariants in each iteration to find appropriate test cases that eliminated all the incorrect invariants output by the program invariant detector. In each case, by the fourth iteration of our method, the program invariant detector inferred only correct invariants. The inference of only correct invariants by the program invariant detector after adding the appropriate test cases suggests that our new test suites had better program specification coverage than our original test suites. Additionally, because the test cases were specifically chosen, the test suites were small while still achieving good coverage.

Most importantly, this research shows the feasibility of using an invariant detector to measure the specification coverage of a test suite and to improve upon that coverage by identifying appropriate tests to add to the suite. An important question that arises, however, is how our metric of measuring specification coverage compares to more common metrics such as branch coverage. We do not attempt to come up with an answer here, but we feel that it is an issue that must be addressed in the future if this is to become common practice.

An issue that could arise while using this method is that the programmer may not have a specification or complete knowledge of the invariants of the program beforehand. While it is possible to deterministically validate or invalidate the invariants produced by a program invariant detector, determining which invariants are missing is a nondeterministic process that depends on the user's knowledge of the invariants. However, we still feel that most programs will benefit from our analysis even if they do not have well defined invariants, or if the developer has not taken the time to manually determine the invariants or to come up with a specification. The reason for this usefulness is that invariants related to critical aspects of the system are certain to be known unless the design is completely unclear. Also, invariants missing invariants from the output are not as important in our analysis as the incorrect invariants. This

focus on incorrect invariants is because incorrect invariants show that there are conditions that have not been tested; if they had been, then these invariants would have been shown to be incorrect. Missing invariants, on the other hand, are generally directly related to the invariant detector. For example, an invariant could be missing if either it is something for which the particular invariant detector used does not test, or if the confidence level of the invariant detector is not high enough to be sure that it is an actual invariant as opposed to a coincidence. Simply adding enough tests so that the invariant detector becomes confident enough for the invariant to be output does not add any quality to the test suite, but instead just adds unneeded tests, which is not the goal of a good test suite.

While we tested our method only on data structures, we feel it would be applicable as well to other user programs. However, we have not determined the applicability of our method to large programs. We suspect that it may not be applicable to larger programs because the invariants can be buried deeper into the code and are often too complex for the type of invariants most dynamic invariant detectors, such as Daikon, are able to detect. Additionally, if the invariants were found for these larger programs, they would likely be more difficult to analyze for correctness.

The scalability issue right now is restricted to the scalability of the invariant detector being used. In our experiments, Daikon was unable to handle large data sets, causing the kernel to kill the process. However, a new beta version of Daikon was released in October, 2003, and its major upgrades from the version we used are in performance and in its ability to handle larger data sets [1]. So, as dynamic program invariant detection improves, the effectiveness of our method should improve.

One critique of our results is that our method evolved slightly as we experimented, so that comparisons of the rate of convergence of incorrect invariants between the `Date` class and the `Polynomial` class cannot be accurately compared. The main change is that when testing for the `Polynomial` class, as we mentioned earlier, we were able to add tests that would prevent new, incorrect invariants of the form  $A! = B$ . This generally would affect only our table for the amount of incorrect invariants found in the second run and should not have changed the total number of iterations before all of the inferred invariants were correct.

Because the intent of our research was to show feasibility, we used a test suite with random inputs, instead of creating the most complete test suite we could. Although in practice this might not be considered a strong test suite, it provided a good base suite that tended to test a good portion of the code while leaving room for improvement. We knew that this would not produce a perfect test suite, but we also knew that if our method suggested a fully tested specification with this suite, then it would be a flawed method.

## IX. CONCLUSION

We have devised a method of improving test suites that is easy to use and has been shown to be effective in determining the coverage in test suites using our specification-based metric. While the method has been shown to be effective and feasible, it is also limited by the performance limitations of the invariant detector which used. In our case, Daikon was limited in the size of programs it can analyze and in the types of invariants it can produce, both of which hinder the capabilities and effectiveness of our method. Future work in this area can address its effectiveness with larger programs as opposed to the relatively small classes we tested in this paper. With the impending release of a newer version of Daikon with the ability to handle larger datasets, the possibilities of using this method on larger programs will be greatly increased. Another possible area of future work is in developing a tool that would automate the process of analyzing the invariants and automatically adding test cases based on them. Also, it would be valuable to compare our new coverage metric with more traditional methods.

## ACKNOWLEDGMENTS

We would like to thank Dr. Coppit for helping us install Daikon. The many hours we spent trying to get Daikon and its components to compile would have been greatly increased had it not been for his generous help.

## REFERENCES

- [1] M. Ernst, "The Daikon invariant detector," URL: <http://pag.lcs.mit.edu/daikon/>.
- [2] W. R. Adrion, M. A. Brandstad, and J. C. Cherniavsky, "Validation, verification, and testing of computer software," *ACM*, vol. 27, June 1982.
- [3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
- [4] M. A. Weiss, "Source code for data structures and problem solving using Java, second edition," URL: <http://www.cs.fiu.edu/~weiss/dsj2/code/code.html>.