Search-Based Failure Discovery using Testability Transformations to Generate Pseudo-Oracles

Phil McMinn
The University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello
Sheffield, UK, S1 4DP
p.mcminn@sheffield.ac.uk

ABSTRACT

Testability transformations are source-to-source program transformations that are designed to improve the testability of a program. This paper introduces a novel approach in which transformations are used to improve testability of a program by generating a *pseudo-oracle*. A pseudo-oracle is an alternative version of a program under test whose output can be compared with the original. Differences in output between the two programs may indicate a fault in the original program. Two transformations are presented. The first can highlight numerical inaccuracies in programs and cumulative roundoff errors, whilst the second may detect the presence of race conditions in multi-threaded code.

Once a pseudo-oracle is generated, techniques are applied from the field of search-based testing to automatically find differences in output between the two versions of the program. The results of an experimental study presented in the paper show that both random testing and genetic algorithms are capable of utilizing the pseudo-oracles to automatically find program failures.

Using genetic algorithms it is possible to explicitly maximize the discrepancies between the original programs and their pseudo-oracles. This allows for the production of test cases where the observable failure is highly pronounced, enabling the tester to establish the seriousness of the underlying fault.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

General Terms

Experimentation, Measurement, Performance

Keywords

Search-based software testing, oracle, pseudo-oracle, non-testable program, program transformation, testability transformation

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

1. INTRODUCTION

Software testing is an extremely costly and laborious process, and as such its automation continues to be a burgeoning interest to software engineering researchers.

Search-based testing [27] is one approach to automating software testing by reformulating the problem as a *fitness function*, the optima of which is sought by optimizing search techniques such as genetic algorithms. Search-based testing has been applied to a number of areas, including functional [5] and non-functional [42] testing, mutation testing [3, 21], exception testing [39], finite state machine testing [11], interaction testing [8], regression testing [44], stress testing [4] and test case prioritization [25, 40]. The majority of work to date, however, has concentrated on automatic test data generation, and structural test data generation in particular [32, 23, 36, 26, 41, 38, 31, 18, 19].

Although it is possible to generate inputs for certain classes of program using search-based techniques, the problem of automatically determining whether the corresponding outputs are correct remains a significant problem; one that is not limited to search-based testing but of major concern for the entire field of software testing. This is because an *oracle*, a mechanism for checking that the output of a program is correct given some input (*e.g.* via a formal specification), is seldom available.

The frequent non-existence of an oracle threatens to undo much of the progress made in automating test data generation, because a human tester is required to perform the task manually. The practical effort in doing this is potentially as great as if the tester had manually generated inputs to the program in the first place. For some programs, it may represent an impossible or highly impractical task. Such programs were referred to by Weyuker as *non-testable* [43].

Davis and Weyuker [10] proposed the use of a 'pseudo-oracle' to alleviate this problem. A pseudo-oracle is a program that has been produced to perform the same task as its original counterpart. The two programs, the original and its pseudo-oracle, are run using the same input and their respective outputs compared. Any discrepancy may represent a failure on the part of the original program or its pseudo-oracle. Traditionally the pseudo-oracle is written independently, perhaps by a different programming team. However, writing programs is itself a costly activity, and producing multiple versions more so.

This paper proposes the automatic generation of pseudo-oracles from a program under test through the use of program transformations. In the context of testing, program transformations used to aid testing or make a program more 'testable' are more commonly referred to as *testability transformations* [16, 17]. The testability transformations proposed in this paper transform an aspect of the program under test into an alternative version, with the intention

of being able to compare outputs from the two versions so as to uncover potential failures in the original program.

Even if a pseudo-oracle can be generated using a testability transformation, it is useless if differences in output cannot be easily found. This paper therefore proposes to utilize the power of meta-heuristic search-based testing techniques to generate inputs that result in such discrepancies. From attempting to maximize differences in output, the tester can be provided with an indication of the potential seriousness of the underlying fault. Since minor inaccuracies may be tolerable for the application concerned, the size of discrepancy found may affect the decision as to whether the underlying fault is one that is worth fixing.

Two pseudo-oracle testability transformations are presented. The first transformation can be used to check programs containing numerical calculations. Numerical calculations and numerical type conversions are a common source of error in computer programs that can go easily undetected during testing. The explosion of the Ariane 5 rocket in 1996 was caused by arguably one of the most high-profile and expensive bugs in history, costing an estimated 370 million US dollars [12]. The on-board control software, included a program statement which assigned a 64-bit floating point value to a 16-bit integer. However, after launch, the floating point variable contained a value too large to be converted to the integer type, the cascading effect of which resulted in the destruction of the craft. In addition to problems of type conversion, the IEEE standard floating point type is notoriously unreliable [13], whilst cumulative rounding errors remain another common source of error in programs that have been exploited for fraudulent financial gain [22] in the past.

The second transformation involves multi-threaded computation. The order of thread execution can potentially have an impact on the integrity of data and the outputs from programs. The transformation imposes serialization constraints on blocks of program statements, therefore any difference between the output of the program under test and the pseudo-oracle produced by this transformation may indicate the presence of a race condition.

The primary contributions of this paper are as as follows:

1. The presentation of two program transformations (testability transformations) for the production of two different types of pseudo-oracle.

2. An approach which uses search-based testing techniques to maximize the difference in output between a program and its pseudo-oracle.

3. The results of an experimental study which applies the search-based approach to find differences between a program and the pseudo-oracles produced by the testability transformations. The results show that both random testing and genetic algorithms are capable of revealing differences in behaviour; whilst the genetic algorithm can maximize the difference further, and hence demonstrate the potential effects of any underlying faults.

This paper is organized as follows. Section 2 provides background information on search-based testing, testability transformations and pseudo-oracles. Section 3 presents two pseudo-oracle producing testability transformations, whilst Section 4 introduces the approach for finding failures using search-based testing techniques. Section 5 presents the results of the experimental study performed using search-based techniques with the pseudo-oracles produced. Section 6 discusses related work whilst Section 7 concludes and relates future work to be undertaken.

2. BACKGROUND

Search-based testing automates a testing activity, such as test data generation, by restating the problem as a *fitness function*, the optimum of which is sought by a meta-heuristic search technique, e.g. a genetic algorithm. Structural test data generation has been by far the most studied form of search-based testing [27]. The search space is formed from the input domain of the program under test, and the goal of the search is to find test data to cover a structure; for example a specific program branch or statement. If, for example, the execution of a specific statement requires a condition in the program `a == b` to be evaluated as true, the fitness function is $|a - b|$. This fitness function is to be minimized, thus, the lower the output of this formula, the closer the values of `a` and `b` are to one another, and the 'closer' the predicate is to being satisfied.

The fitness function provides the search with a sense of 'direction' for navigating a potentially very large input domain. If for example `a = 5` and `b = 10`, the search knows by virtue of fitness values assigned to input vectors that a small increase of `a` is a 'good' move, which takes it closer to the optima and the required test data; whereas increasing `b` is a 'bad' move, because the resulting fitness value is worse.

For certain types of program structure, the fitness function does not offer any guidance to search. For example, when a boolean 'flag' variable is a branching condition, there are only two fitness values representing the entire input domain; one value where the flag is true, and the other where the flag is false. With such coarse fitness information a meta-heuristic search will become random. Harman *et al.* [16, 17] introduced a type of program transformation called a *testability transformation* to address this problem. The purpose of a testability transformation is to change a program in order to make it more 'testable', in this case easing the process of automatic test data generation by removing the flags that are obstructing the search process. In the transformed version of the program, the flag is replaced by the conditions that lead to the flag becoming true or false. When the fitness function is based on these flag-free conditions, useful fitness information can be retrieved for guiding the search to the required inputs. Once test data has been successfully generated, the transformed program is discarded; its purpose as an intermediary for improving the testing process served.

Although search-based testing can generate test data, an oracle is seldom available. An oracle is used to verify that the outputs produced by the inputs are the correct ones. A human is required to perform the role, a task which is almost as time consuming and costly as generating test data in the first place. For some programs this is almost an impossible task, programs which Weyuker labelled 'non-testable' [43]. Davis and Weyuker suggested the creation of a pseudo-oracle, an alternative program written to perform the same task. The alternative program is *pseudo* in the sense that it does not guarantee that the outputs of the original program are 'wrong', but where the pseudo-oracle and the original program differ, the causes should at least be investigated further, because at least one of the programs must be faulty.

Writing a program multiple times is itself costly. This paper introduces testability transformations to automatically generate pseudo-oracles from certain classes of program. Search-based techniques are then used to try and generate inputs that cause the program and its pseudo-oracle to give different outputs.

3. PSEUDO-ORACLE GENERATION VIA TESTABILITY TRANSFORMATION

This section details two testability transformations used in this paper to generate two types of pseudo-oracle.

	Using the `double` primitive type	Using the `java.math.BigDecimal` class

Java statement:	`System.out.println(0.1 + 0.1 + 0.1);`	`System.out.println(new BigDecimal("0.1").add(`
		`new BigDecimal("0.1")).add(`
		`new BigDecimal("0.1")));`

Output:	`0.30000000000000004`	`0.3`

Figure 1: Comparing floating-point arithmetic in Java (version 6) using `double` compared to `BigDecimal`

3.1 Numerical calculations

Numerical errors can subtly manifest themselves into programs in a variety of ways with potentially disastrous consequences. In Java, the `int` and `long` integer types are subject to silent overflow errors [20]. The special error values `INF` (infinity) and `NaN` ('not a number'), returned by floating-point operations, can also propagate through a program without exceptions being thrown. Further problems can arise as a result of using floating-point representations conforming to the IEEE standard, used by both C and Java, which is incapable of representing certain numbers of finite decimal representation, such as `0.1` [13]. For example in Java, the operation `0.1 + 0.1 + 0.1` yields '`0.30000000000000004`' rather than simply `0.3` (Figure 1). If allowed, resulting inaccuracies can accumulate into serious discrepancies during the course of the program.

Small numerical errors can also accumulate into larger errors through premature rounding. This type of fault is present in the class of Figure 2a, which is intended to represent a bank account[1]. The programmer is aware of the perils of representing the amount in the account using a `double` variable, and instead decides to use the `long` integer type to represent the amount in the account as pennies. The problems begin when interest is added to the account. Values added in the `addInterest` method are silently cast back to the `long` type of the `amount` variable, with fractional penny amounts inadvertently rounded down in the process. These fractional differences may have the long term effect of a customer losing out on compound interest. For example if £1,000 were deposited at an interest rate of 25%, the account is a penny down after three of applications of `addInterest`, and over £2 out after the 25th. Despite the amount not being stored as a `double`, the class still suffers from the occasional inaccuracies resulting from the intermediate implicit `double` calculation involving the `interestRate` variable in the `addInterest` method.

The Convert-to-BigDecimal Transformation

The `BigDecimal` class in Java, which resides in the `java.math` package [46], can alleviate some of the above problems, and is the basis for the pseudo-oracle transformation proposed here. `BigDecimal` objects are capable of accurately representing numbers of finite decimal representation with arbitrary precision. Arithmetic is performed by using class methods (*e.g.* the `add`, `subtract` and `multiply` methods). When faced with illegal operations, the class will throw exceptions rather than using special error values, which will terminate the program (unless handled appropriately).

The cost behind using `BigDecimal` is verbose code (Figure 1) and a more inefficient program. The verbosity of code in particular presents readability and inevitable maintenance issues. Thus, `BigDecimal` may not be a natural choice if speed or ease of readability and maintenance is more important than program accuracy. Or, the developers may believe that accuracy is not an issue; and thus may be unaware of potential and possibly very subtle faults in their code that may be brought to their attention by tests using a pseudo-oracle. Of course, maintenance and code readability are not issues for testability transformations, which are automatically generated and thrown away when testing is complete.

The *Convert-to-BigDecimal* transformation produces a pseudo-oracle for unit testing by taking a Java class and replacing variables of primitive numerical types with instances of the `BigDecimal` class. Operations involving the original variables (*e.g.* +, -, * and so on) are replaced with the appropriate method invocations (*i.e.* `add`, `subtract`, `multiply` *etc.*) on the `BigDecimal` object.

In accordance with these rules, the `BankAccount` class of Figure 2a can be automatically transformed to the class of Figure 2b, `BankAccount_BigDecimal`, a more accurate version that will be used in the experimental study presented in Section 5 as a pseudo-oracle.

3.2 Race Conditions

A race condition exists in a piece of code when the order of execution of two or more threads affects the value of a variable in a program or an output.

In the `BankAccount` example, a race condition exists on the value of the `amount` instance variable[2]. Suppose the account has £500 in it. An execution thread attempts to withdraw £400, satisfying the '`amount > withdraw`' condition appearing in the `if` statement of the `withdraw` method. Before the condition is fully evaluated, however, a second execution thread invocates the method with an instruction to withdraw a further £400. Since the first thread has not actually deducted its amount from the account, the second thread also passes the same condition, as at this point there are still adequate funds in the account. Both threads are then free to continue to execute the true branch of the `if` statement, withdrawing a combined figure of £800. This leaves the account overdrawn, and in an illegal state, since the class is programmed such that the account is clearly not supposed to be in deficit.

The solution to this problem is to serialize access to the method involving the race condition, so that a thread can not execute it at the same time as another. This is achieved using the `synchronized` keyword in Java (Figure 2c). This is the basis of the *Add-Synchronization* pseudo-oracle transformation.

Add/Remove-Synchronization Transformation

The *Add-Synchronization* pseudo-oracle transformation simply takes a class and adds the `synchronization` keyword to its methods. In Figure 2, the `BankAccount` class (part a of the figure) is transformed to the `BankAccount_Synchronized` class of part c. If the pseudo-oracle produces a different result to the original in the presence of threaded code, a race condition may be present.

The inverse of this transformation (*remove-synchronization*) may also be useful in determining whether the synchronization points added to a class are strictly necessary. Synchronization is an inherent bottleneck in code, since threads must wait for other threads to release their lock on an object, method or block in the code. Synchronization can also be responsible for deadlocking issues. A *remove-synchronization* transformation may help determine whether these locks are actually needed in practice. A further usage of the *add/remove-synchronization* transformation may also be the dynamic validation of the results of static model checkers.

[1] Example adapted from Sedgewick and Wayne [37]

[2] Example adapted from Oaks and Wong [33]

```
public class BankAccount {

    private long amount;
    private int interestRate;

    public BankAccount() {
        amount = 0;
        interestRate = 0;
    }

    public void deposit(long depositAmount) {
        amount += depositAmount;
    }

    public void withdraw(long withdrawalAmount) {
        if (amount > withdrawalAmount)
            amount -= withdrawalAmount;
    }

    public void addInterest() {
        amount *= (1 + interestRate / 100.0);
    }

    public void setInterestRate(int newRate) {
        if (newRate >= 0 && newRate <= 25)
            interestRate = newRate;
    }

    public int getInterestRate() {
        return interestRate;
    }

    public long getAmount() {
        return amount;
    }
}
```

(a) Original version

```
import java.math.BigDecimal;

public class BankAccount_BigDecimal {

    private BigDecimal amount;
    private BigDecimal interestRate;

    public BankAccount_BigDecimal() {
        amount = new BigDecimal("0");
        interestRate = new BigDecimal("0");
    }

    public void deposit(BigDecimal depositAmount) {
        amount = amount.add(depositAmount);
    }

    public void withdraw(BigDecimal withdrawalAmount) {
        if (amount.compareTo(withdrawalAmount) > 0)
            amount = amount.subtract(withdrawalAmount);
    }

    public void addInterest() {
        amount = amount.multiply(interestRate.divide(
                  new BigDecimal("100")).add(new BigDecimal("1")));
    }

    public void setInterestRate(BigDecimal newRate) {
        if (newRate.compareTo(new BigDecimal("0")) >= 0
                && newRate.compareTo(new BigDecimal("25")) <= 0)
            interestRate = newRate;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public BigDecimal getAmount() {
        return amount;
    }
}
```

(b) *Convert-to-BigDecimal* **transformation**

```
public class BankAccount_Synchronized {

    private long amount;
    private int interestRate;

    public BankAccount_Synchronized() {
        amount = 0;
        interestRate = 0;
    }

    public synchronized void deposit(long depositAmount) {
        amount += depositAmount;
    }

    public synchronized void withdraw(long withdrawalAmount) {
        if (amount > withdrawalAmount)
            amount -= withdrawalAmount;
    }

    public synchronized void addInterest() {
        amount *= (1 + interestRate / 100.0);
    }

    public synchronized void setInterestRate(int newRate) {
        if (newRate >= 0 && newRate <= 25)
            interestRate = newRate;
    }

    public synchronized int getInterestRate() {
        return interestRate;
    }

    public synchronized long getAmount() {
        return amount;
    }
}
```

(c) *Add-synchonization* **transformation**

Figure 2: The `BankAccount` Java class and two pseudo-oracles generated by two different testability transformations

(a) The original version of the class. Aware of problems with the `double` type, the programmer uses the `long` type to store amounts in pennies. This can lead to a cumulative rounding error when interest is calculated

(b) Pseudo-oracle produced by the *Convert-to-BigDecimal* transformation. The pseudo-oracle uses `BigDecimal` for the instance variables of the class, and is capable of exposing rounding error problems

(c) Pseudo-oracle produced by the *Add-Synchonization* transformation. In the pseudo-oracle all methods are marked as `synchronized`, enabling exposure of race conditions where the class is used in a multi-threaded environment

4. FAILURE DISCOVERY VIA SEARCH-BASED TECHNIQUES

Using the pseudo-oracles, this paper proposes to apply search-based testing with the intent of answering the following questions:

1. **Does a fault exist?** When executed with the same input, does the output from the original and its pseudo-oracle transformation diverge? If yes, the discrepancy could represent a failure in the original program and thus the presence of a fault.

2. **If so, how severe is it?** Whilst discovering a failure is generally sufficient for a test to be deemed successful, severity of the fault is also an important concern in practice. Any indication of fault severity may help prioritize bug fixes that may need to be carried out. If the failure that results from the fault is relatively minor, the bug might not even be worth fixing. Some degree of numerical inaccuracy, for example, may be tolerable for the application concerned. Search-based testing in combination with a pseudo-oracle may be able to assist in answering questions of severity by maximizing differences in output between the two versions of a program.

Existing structural test data generation techniques could be used in attempt to uncover differences, however it is unlikely that the resulting test data would be able to 'maximize' the effect of a potential fault, and thus demonstrate its severity. For example, cumulative rounding implies a certain amount of repetition, yet branch coverage test data is generated for a loop if it is entered once or not at all. Likewise, race conditions are not represented by structures that the search must attempt to 'cover'.

Therefore, in this paper, the search attempts to generate test data that attempts to expose the maximum difference in output between the two programs in order to demonstrate fault severity (consequently allowing a programmer to decide if the fault is worth fixing). The fitness function is responsible for evaluating how 'different' the outputs are. If the outputs are identical for some input, its fitness value is zero.

The pseudo-oracle transformations presented in this paper operate at the level of a Java class, 'output' is considered to be the values returned by instance methods of the class. Therefore, difference in output is considered to be the difference in the return values of 'accessor' methods for an object of the original class and the transformed version of the class, given an identical method call sequence and associated parameter values. In accordance with this, fitness is computed using the following formula for the `BankAccount` example of Figure 2:

$$fitness = |orac.getAmount() - orig.getAmount()| + |orac.getInterestRate() - orig.getInterestRate()| \quad \textbf{(1)}$$

where *orac* is an instance of the transformed pseudo-oracle class and *orig* is an instance of the original `BankAccount` class.

To the casual observer, it may seem a strange choice to include all accessor methods in the fitness function for `BankAccount`. `getInterestRate`, for example, will not have an effect on fitness with the *Convert-to-BigDecimal* transformation. However, in general, this would not be known *a priori*.

The search space is not just the input domain of an individual method or function, as is usually the case in search-based testing [36, 26, 41, 18, 19], but rather sequences of calls to objects that are required for testing the class, such as those generated in the object-oriented approach of Tonella [38]. The scheme applied for experiments in this paper is detailed in the next section.

5. EXPERIMENTAL STUDY

The experimental study took the `BankAccount` example presented in Figure 2a, and applied random search and a genetic algorithm to find method call sequences which maximized the fitness function of Equation 1. The call sequences are applied first to an object constructed from the class under test (*i.e.* `BankAccount`), and then to an object of its transformed pseudo-oracle class (*i.e.* `BankAccount_BigDecimal` or `BankAccount_Synchronized`).

The exact representation of the call sequence optimized by the search methods is specific to each pseudo-oracle (non-threaded and threaded) and is detailed in the sections below.

The random search simply generated 10,000 call sequences at random. The genetic algorithm used a population size of 100, and tournament selection for reproduction with a tournament size of 2. Discrete recombination was used for crossover, with Gaussian mutation applied at a rate $p_m = \frac{1}{len}$ (where *len* refers to the length of the individual's chromosome). Reinsertion was applied using an elitist strategy; with 90% of the least-fit individuals of the previous generation replaced with the best 90% of the offspring.

Both algorithms were terminated after 10,000 call sequence evaluations, *i.e.* 10,000 executions of the original program and its transformed pseudo-oracle counterpart. Each experiment performed was repeated 20 times using distinct random seeds.

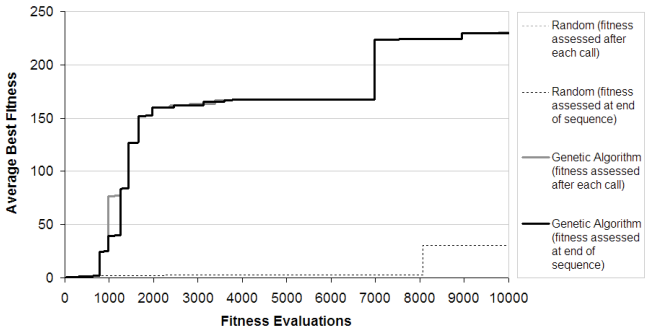
5.1 Convert-to-BigDecimal Transformation

Four versions of this experiment were carried out using each search method. Two fixed call sequence sizes were optimized by each search; a length of 10 and a length of 20. Each individual method call in the sequence is represented as a pair of integer numbers (m, p), where $0 \leq m \leq 3$ is an identification number that refers to one of the four methods to be called (the `getInterestRate` and `getAmount` methods, which do not mutate the state, were not included), whilst p, $0 \leq p \leq 1000$, represents an optional parameter to the method. In addition, two versions of fitness assessment were used. In the first, the fitness function was computed after each method call to the program under test and its pseudo-oracle, with the maximum value used as the final fitness value for the entire call sequence. In the second, the fitness function was computed once, at the end of the sequence.

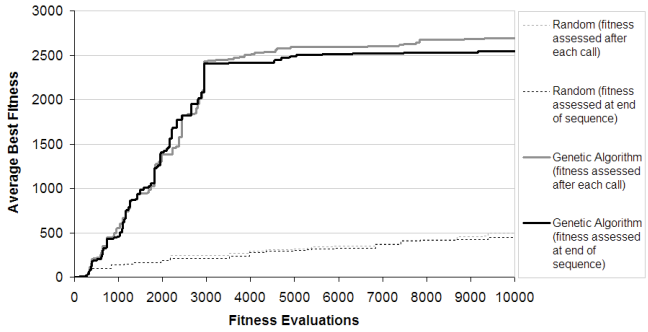
Figure 3a and 3b plot search progress for call sequence lengths of 10 and 20 respectively. The graphs plot 'average best fitness' against the number of fitness evaluations. Average best fitness is simply the average of the best fitness values obtained by each of the 20 trials of the search technique at a given point in the search process. The plots reveal that the decision as to where fitness should be computed had very little bearing on the end result, with the same type of search (random or genetic algorithm) performing in a similar fashion. In all cases, the genetic algorithm outperformed random search, and perhaps unsurprisingly the longer call sequence allowed for higher fitness values (*i.e.* higher numerical discrepancies) to be found for both types of search.

Table 1 shows the maximum, minimum and average fitness values obtained by the end of each search over each of the 20 trials, along with their standard deviation. The table shows a large difference in the maximum and minimum fitness values over the 20 trials and a high standard deviation. It would therefore seem that the search landscape may not have been particularly smooth, with both searches making progress through sequences of 'fortuitous' moves that allowed them to hop between local optima or navigate off plateaux.

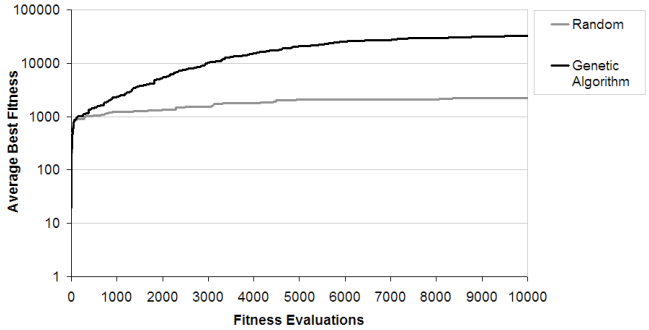
Overall, the maximum fitness found by the genetic algorithm was 11,317. The `amount` instance variable is the single source of numerical error in the class, meaning that the search was able to



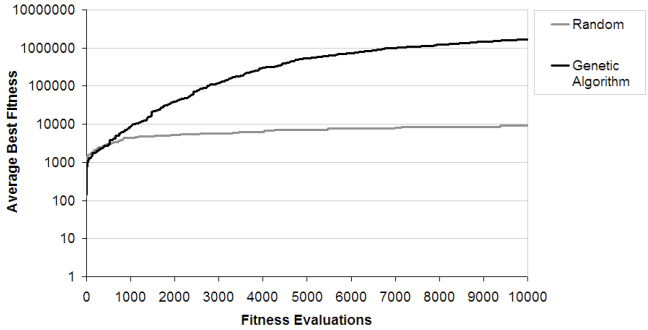
(a) *Convert-to-BigDecimal* **pseudo-oracle, sequence length = 10 (The lines for random search appear one on top of the other)**



(b) *Convert-to-BigDecimal* **pseudo-oracle, sequence length = 20**



(c) *Add-synchronization* **pseudo-oracle, sequence length = 10**



(d) *Add-synchronization* **pseudo-oracle, sequence length = 20**

Figure 3: Average best fitness over each set of 20 trails involving each pseudo-oracle transformation and experimental setup

uncover a discrepancy of £113 between accounts represented by the original class and the pseudo-oracle. The sequence that the genetic algorithm found achieved this large discrepancy by interesting means. In essence, a small error (a fraction of a penny) is initially induced between the two versions. The sequence begins by making a deposit of 701p, setting the interest rate set to 22%, and then adding interest. The transformed version now has 855.22p whilst the original has exactly 855p. Next, a method call is made to withdraw all the money from the account. That is, the `withdraw` method is invoked with a parameter that is equal to the number of complete pennies (855) in the account. Because the transformed version has a fraction of a penny more in the account, the amount in the account exceeds the withdrawal amount, and so the request is granted. In the original class, however, the amount in the account is exactly the same as the withdrawal amount, and the request is not granted. The two accounts now differ by 855p - the original having 855p exactly and the transformed version having 0.22p. Further method calls in the sequence then call `addInterest`. These calls will not result in much interest being added to the amount in the transformed version, with the amount increasing rapidly for the original version, resulting in a final difference of £113.

5.2 Add-Synchronization Transformation

In order to experiment with the *Add-Synchronization* transformation, two execution threads were created to act simultaneously on the objects created from the original `BankAccount` class and the `BankAccount_Synchronized` pseudo-oracle.

Two fixed call sequence sizes were used, one of 10 method calls per thread, and another of 20 calls per thread. Each method call is represented as a triple (m, p, w), with m and p corresponding to a method number and a parameter as for the *Convert-to-BigDecimal* experiment; whilst w represents a number of milliseconds the thread should wait before proceeding to the next call. The overall sequence is arranged such that the first half corresponded to method calls for the first thread, with the second half reserved for the second thread. The two threads executed their call sequences simultaneously, with both first acting on the instance of `BankAccount` and then on the instance of `BankAccount_Synchronized`. As calls to the program under test and pseudo-oracle were not performed in a discrete fashion, as for the *Convert-to-BigDecimal* experiment, fitness was solely measured at the end of each call sequence.

Figure 3c and 3d plot search progress on a logarithmic scale, with the genetic algorithm clearly outperforming random search. Significant differences in output were found, as further recorded through maximum fitness values reported in Table 1. Again, a high variance was found amongst the 20 repetitions of the experiments, perhaps indicating again that the search landscape was far from smooth, but nonetheless conducive to the discovery of large discrepancies, particularly in the case of the genetic algorithm.

5.3 Conclusions from the Experimental Study

The results of the above experiments show that search-based techniques are indeed capable of finding differences in output between a program under test and the pseudo-oracle transformations detailed in this paper, revealing failures caused by the faults present in the `BankAccount` class, which were originally discussed in Section 3. The failures could also be accentuated by the genetic algorithm. This may provide the programmer with an indication of fault severity. This is useful information when a degree of inaccuracy may be tolerable for the application concerned, or, bug fixes need to be prioritized. Given the variance of the results, it would seem that the fitness landscape is not particularly smooth, with 'fortuitous' jumps sometimes required for the search to make progress. This is an issue for future work.

Table 1: Minimum, maximum and average fitness found using each pseudo-oracle transformation in each experimental setup

	Sequence length = 10				Sequence length = 20			
	Min	Max	Average	Standard Deviation	Min	Max	Average	Standard Deviation
Pseudo-oracle generated by the *Convert-to-BigDecimal* transformation								
Fitness assessed after each call								
Random	2	554	31	120	9	970	500	344
Genetic Algorithm	3	1,439	230	420	39	11,317	2,690	344
Fitness assessed at end of sequence								
Random	2	554	31	120	9	966	450	344
Genetic Algorithm	3	1,436	230	420	39	11,317	2,541	344
Pseudo-oracle generated by the *Add-Synchronization* transformation								
Random	1,420	3,835	2,228	717	2,914	19,386	9,144	4,158
Genetic Algorithm	4,340	75,020	32,763	18,039	71,408	5,434,587	1,699,271	1,269,800

6. RELATED WORK

The idea of a pseudo-oracle was first introduced by Davis and Weyuker [10] as a means of tackling *non-testable* programs [43], *i.e.* those without an oracle or involving insurmountable difficulty in checking the correctness of outputs. The pseudo-oracle was intended to be an alternative version of the program produced independently, *e.g.* by a different programming team or written in a different programming language. The premise of producing multiple versions of a program was used prior to this in fault-tolerant computing, where it was referred to as *multi-* or *N*-versioning [1, 2]. In this approach a highly critical piece of software would be implemented in multiple ways and executed in parallel. Where the outputs differed, a 'voting' mechanism was proposed to decide which output would be used.

Metamorphic testing [6, 45] is an alternative approach to the oracle problem that does not involve multiple implementations of a program. In metamorphic testing, future outputs are predicted using knowledge of previous outputs and the use of a series of *metamorphic relations* which are derived from properties inherent in the program under test. Metamorphic testing differs from the approach presented in this paper, since it is past outputs from the program that are 'transformed', as opposed to the program itself. Clark and Hierons have investigated the possibility of combining metamorphic testing with search-based approaches [7].

It is interesting to relate the idea of producing a pseudo-oracle via program transformation to mutation testing [34]. In some aspects the transformations presented in this paper could be viewed as higher-order mutants [21]. However, the intention behind using transformations to generate pseudo-oracles is not ostensibly to introduce faults, as the pseudo-oracle is intended to fulfill the same specification as the original. Conversely, the pseudo-oracle is not intended to be an equivalent mutant [35] either. If equivalence were always preserved, the technique would be useless, as there would never be a difference in output, and failures would never be discovered.

The idea of using a program transformation to improve the testability of a piece of code was first introduced by Harman *et al.* [16, 17]. Testability transformations have traditionally been used to improve test data generation [15]. The transformation replaces the program under test, but the test data produced is still adequate for the original program. Recently McMinn *et al.* [30, 29] and Korel *et al.* [24] have proposed different types of testability transformations categorized as *probabilistic* transformations by Harman [14]. Here, the transformation is not guaranteed to improve the test data generation process, but may nevertheless do so in many situations, as shown through empirical studies.

In a previous workshop abstract [28], McMinn proposed a probabilistic form of transformation called a *co*-testability transformation, which is intended to work in *conjunction* with the original program rather than instead of it (and in conjunction with other transformations if available). The pseudo-oracles generated in the present paper are a type of co-testability transformation. The co-testability transformation abstract presented a variant of the *Convert-to-BigDecimal* transformation. For the particular examples discussed in the abstract, random search and genetic algorithms were able to uncover discrepancies in program output between the original and transformed version of the program, however genetic algorithms were unable to maximize discrepancies any more than random search due to the choppy and flat nature of the fitness landscapes for the particular programs involved.

The initial ideas behind search-based testing emerged in the 70s, with the seminal work of Miller and Spooner who used numerical maximization techniques to generate test data for path coverage [32]. These ideas went largely ignored until Korel developed them further in 1990 [23]. Since then fitness functions have been developed for different types of structural coverage, *e.g.* branch coverage [36, 26, 41], different types of programming paradigm, *e.g.* object-oriented languages [38] as well as different forms of search technique, such as genetic algorithms [36, 26, 41, 38, 18, 19]. Search-based testing has also been applied to many other forms of testing [5, 42, 3, 21, 39, 11, 8, 44, 4, 25, 40]. This is the first work, however, that has applied search-based techniques to discover failures using pseudo oracles. It is also the first work that has used search-based testing techniques as a differencing tool in order to discover where the output from two programs varies.

7. CONCLUSIONS AND FUTURE WORK

This paper has introduced testability transformations for generating pseudo-oracles. Pseudo-oracles can be used in combination with search-based testing techniques to automatically find failures and maximize the effects of an underlying fault in a program under test. 'Maximization' of a failure helps give an indication of fault severity, since a degree of inaccuracy may be tolerable for the application concerned. Severity information is also useful for prioritizing work on bug fixes.

An experimental study showed the effectiveness of the approach, in which random testing and a genetic algorithm were able to highlight failures of a numerical and synchronous nature. With respect to maximizing fault severity, the best results were obtained with the genetic algorithm. The results imply, however, that the fitness landscapes produced by the fitness function were not smooth. Improvements may be made by incorporating branch distance calculations

used in search-based structural testing into the fitness function [27], so as to guide the search to the execution of individual statements that have been especially transformed for generation of the pseudo-oracle. This may allow variations in output to be found in a more effective fashion.

Future work also intends to discover new forms of pseudo-oracle that can be generated using testability transformations. The use of search-based techniques as a tool to discover differences in behaviour between two different versions of a program may also have further applications; for example the evaluation of different programming approaches on non-functional requirements (such as timing, heat generation or power usage in an embedded controller); evaluating migration strategies and different choices of API; or improved testing of the integrity of automated refactorings [9].

8. ACKNOWLEDGMENTS

The author would like to thank John Clark, Mark Harman, Rob Hierons, Gregory Kapfhammer, Kiran Lakhotia, Marc Roper, Chris Thomson and Shmuel Ur for useful comments and discussions regarding this work.

Phil McMinn is supported in part by EPSRC grant EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EPSRC grant EP/F065825/1 (REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference).

9. REFERENCES

[1] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Trans. Software Engineering*, 11:1491–1501, 1985.

[2] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault-tolerance during execution. *First International Computer Software and Application Conference (COMPSAC '77)*, pp. 149–155, 1977.

[3] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification & Reliability*, 15:73–96, 2005.

[4] L. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. *Genetic and Evolutionary Computation Conference (GECCO '05)*, pp. 1021–1028, Washington DC, USA, 2005. ACM Press.

[5] O. Buehler and J. Wegener. Evolutionary functional testing of an automated parking system. *International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, USA, 2003.

[6] T. Chen, S. 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.

[7] J. Clark and R. Hierons. Search-based metamorphic testing. Technical report, University of York, unpublished, 2003.

[8] M. Cohen, P. Gibbons, W. Mugridge, and C. Colbourn. Constructing test suites for interaction testing. *25th International Conference on Software Engineering (ICSE '03)*, pp. 38–48. IEEE Computer Society, 2003.

[9] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, pp. 185–194, 2007.

[10] M. Davies and E. Weyuker. Pseudo-oracles for non-testable programs. *ACM '81 Conference*, pp. 254–257, 1981.

[11] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of FSMs. *The Computer Journal*, 39:331–344, 2006.

[12] M. Dowson. The Ariane 5 software failure. *Software Engineering Notes*, 22:84, 1997.

[13] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 21(1):5–48, 1991.

[14] M. Harman. Open problems in testability transformation. *1st International Workshop on Search-Based Testing*. IEEE digital library, 2008.

[15] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation - program transformation to improve testability. *Formal Methods and Testing, Lecture Notes in Computer Science*, volume 4949, pp. 320–344. Springer-Verlag, 2008.

[16] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. *Genetic and Evolutionary Computation Conference (GECCO '02)*, pp. 1359–1366, New York, USA, 2002. Morgan Kaufmann.

[17] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Software Engineering*, 30(1):3–16, 2004.

[18] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. *International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 73–83, London, UK, 2007. ACM Press.

[19] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: local, global and hybrid search. *IEEE Trans. Software Engineering*, to appear, 2009.

[20] Bug ID: 4466549 Add saturation/overflow integer arithmetic. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4466549.

[21] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. *8th International Working Conference on Source Code Analysis and Manipulation (SCAM '08)*, pp. 249–258, Beijing, China, 2008. IEEE Computer Society.

[22] M. E. Kabay. Salami fraud. Network World Security Newsletter, http://www.networkworld.com/newsletters/sec/2002/01467137.html, 2002.

[23] B. Korel. Automated software test data generation. *IEEE Trans. Software Engineering*, 16(8):870–879, 1990.

[24] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta and Q. Zhang. Data dependence based testability transformation in automated test generation. *16th International Symposium on Software Reliability Engineering (ISSRE '05)*, pp. 245–254, Chicago, Illinios, USA, 2005.

[25] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Software Engineering*, 33:225–237, 2007.

[26] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Software Engineering*, 27(12):1085–1110, 2001.

[27] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[28] P. McMinn. Co-testability transformation. *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART '08), Fast Abstract*, 2008.

[29] P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. *UK Software Testing Workshop (UKTest '05)*, pp. 165–182. University of Sheffield Computer Science Technical Report CS-05-07, 2005.

[30] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, 18(3), 2009.

[31] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. *International Symposium on Software Testing and Analysis (ISSTA '06)*, pp. 13–24, Portland, Maine, USA, 2006. ACM Press.

[32] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Engineering*, 2(3):223–226, 1976.

[33] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 2nd edition, 1999.

[34] A. J. Offutt. A practical system for mutation testing: Help for the common programmer. *International Test Conference*, pp. 824–830, 1994.

[35] A. J Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification & Reliability*, 7:137–194, 1997.

[36] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[37] R. Sedgewick and K. Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison Wesley, 1st edition, 2007.

[38] P. Tonella. Evolutionary testing of classes. *International Symposium on Software Testing and Analysis*, pp. 119–128, Boston, USA, 2004. ACM Press.

[39] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

[40] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time aware test suite prioritization. *International Symposium on Software Testing and Analysis (ISSTA '06)*, pp. 1–12. ACM Press, 2006.

[41] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[42] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, 2001.

[43] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25, 1982.

[44] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. *International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 140–150, New York, NY, USA, 2007. ACM Press.

[45] Z. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen. Metamorphic testing and its applications. *8th International Symposium on Future Software Technology (ISFST '04)*, 2004.

[46] J. Zukowski. The need for BigDecimal. Core Java Technologies Tech. Tips, http://blogs.sun.com/CoreJavaTechTips/entry/the_need_for_bigdecimal, 2007.