

Co-testability Transformation

Phil McMinn

Department of Computer Science, University of Sheffield
211 Portobello, Sheffield, S1 4DP, UK

Abstract

This paper introduces the notion of ‘co-testability transformation’. As opposed to traditional testability transformations, which replace the original program in testing, co-testability transformations are designed to be used in conjunction with the original program (and any additional co-transformations as well). Until now, testability transformations have only been used to improve test data generation. However, co-testability transformations can function as partial oracles. This paper demonstrates practical usage of a co-testability transformation for automatically detecting floating-point errors in program code.

1 Introduction

The original purpose of a *testability transformation* was to change a program so that test data could be generated more easily [2, 3, 4, 8, 9]. Traditionally, the transformed program replaced the original program in the test data generation process, typically removing some feature in the code that presents an obstacle for the test data generator.

This paper introduces the notion of a *co-testability transformation*, which is designed to be used in *conjunction* with the original program. The idea opens up the possibility of using multiple transformations at once in order to leverage different ‘tactics’ to coerce testability, or as a vehicle for supplying the test data generation process with additional information about test goals.

This paper, however, introduces the novel use of transformation as a means of providing a *partial oracle*. The paper demonstrates this idea through the automatic detection of errors in code with floating-point calculations. The co-transformation uses floating-point representations and operators of a higher precision. Search-based testing [7] is used to maximize the difference in output between the co-transformation and the original program, thus alerting the tester to errors that may be lurking in the original program.

2 Co-testability transformation oracles for code with floating-point calculations

Floating-point calculations are a common source of error in computer programs that can go easily undetected. Not

all numbers can be represented with complete accuracy on a computer and some floating-point operations cannot be faithfully computed in all circumstances. Subtle rounding-errors can also be introduced which can accumulate into serious discrepancies during the course of the program. The IEEE standard for floating-point numbers, used by C and Java, introduces additional problems [1]. It is incapable, for example, of representing certain numbers of finite decimal representation (e.g. 0.1) and uses error values INF (infinity) and NaN (‘not a number’) in arithmetic operators which can be free to silently propagate through a program.

Figure 1a serves to demonstrate some of these problems. The quadratic equation $1x^2 + 2x + 3 = 0$ has no real roots because the discriminant $b^2 - 4ac$ is negative. However, the operation `Math.sqrt(d)` returns NaN, which is then used by subsequent statements in the program. Also, for the equation $2x^2 + 0.1x - 3$, the function reports 1.200000023841858 and -1.2500000238418578 as roots, which should be 1.2 and -1.25 precisely.

The single co-transformation used for the problem (Figure 1b) involves taking the original Java method and changing variables of `double` or `float` type to that of type `java.math.BigDecimal` [10], which is capable of accurately representing floating-point numbers of finite decimal representation with arbitrary precision. Operations involving floating-point variables in the original program are translated to the corresponding operators for `BigDecimal` (the `Root.square` operation was specially imported from a third-party library). The transformed version is capable of producing results of higher accuracy than the original, and will throw exceptions when faced with illegal operations rather than using IEEE error values. Such exceptions will crash the program if unchecked and thus will not go unnoticed.

Whilst having the obvious benefits of producing more accurate results, the transformed version of the program is more verbose than the original and is likely to be more inefficient in terms of computing resources. Such factors could be long-term concerns for maintenance and live operation of the system, which is why the original version of the program may not have been written this way. However, such factors are of little concern for a comparatively short-term

testing process, after which the transformed version of the program can be discarded (as is the case with all types of testability transformation).

The two versions of the program can be checked against one another for behavioural differences using search-based testing. The fitness function used simply seeks to maximize differences in output between the original and transformed versions of the program when executing using the same inputs. The search is therefore able to trigger exceptions and highlight the point of greatest inaccuracy in floating point computation. A certain degree of numerical inaccuracy may be tolerable for the application concerned, and so the tester is required to weigh up where the results discovered by the search process constitute a serious problem.

3 Initial Experiments and Results

Experiments were performed using the quadratic solver example of Figure 1 and the tax calculation ('calcTax') example of Figure 2 (adapted from [10]). Random, hill climbing and evolutionary searches (configured as described in [5]) were repeated 30 times, and the maximum absolute error difference found between the two versions of the program over the searches was averaged.

For calcTax with a domain size of 10^9 each search was able to reveal a similar discrepancy of 0.99 - 1.00 (i.e. almost a complete unit of currency) in the calculated result. One would normally expect random search to perform poorly compared to hill climbing and evolutionary search, but this was not the case, with little variation exhibited across the search techniques. This would seem to indicate that either random search is sufficient to reveal differences between the original and transformed version of the program, or the fitness landscape did not provide sufficient guidance for the heuristic searches to make any meaningful progress.

For quadSolve each search had trouble *not* testing with inputs that corresponded to a quadratic equation with imaginary roots, and thus causing the transformed program to throw an exception and not return a result. Whilst it was good that this discrepancy was easily discovered, it was clear the program needed to be transformed in an alternate way or the fitness function needed more information to actually produce an output for comparison purposes, e.g. the addition of a distance calculation to avoid the square root operation being called with a negative number. This is an issue for future work.

4 Future Work

Future work will further develop the co-testability transformation concept. The idea of using multiple transformations rather than just one in combination with the original program is appealing and may fit well with competing sub-population, co-evolutionary or multi-objective approaches

<pre>void quadSolve(double a, double b, double c) { double d = b*b - 4*a*c; /* discriminant */ double sqrtD = Math.sqrt(d); double x1 = (-b + sqrtD) / (2 * a); double x2 = (-b - sqrtD) / (2 * a); System.out.println("The roots are "+x1+" and "+x2); }</pre>
(a) Original Java method (adapted from [6])
<pre>void quadSolve(BigDecimal a, BigDecimal b, BigDecimal c) { BigDecimal d = (b.multiply(b)).subtract(new BigDecimal(4).multiply(a).multiply(c)); BigDecimal sqrtD = Root.square(d); BigDecimal x1 = (b.negate().add(sqrtD)). divide(a.multiply(new BigDecimal(2))); BigDecimal x2 = (b.negate().subtract(sqrtD)). divide(a.multiply(new BigDecimal(2))); System.out.println("The roots are "+x1+" and "+x2); }</pre>
(b) Co-testability transformation oracle

Figure 1. Quadratic equation solver example

```
double calcTax(double amount, double discountPercentage,
    double taxPercentage) {
    double discount = amount * discountPercentage;
    double total = amount - discount;
    double tax = total * taxPercentage;
    double taxedTotal = tax + total;
    return Math.round(taxedTotal * 100) / 100;
}
```

Figure 2. calcTax example

for search-based testing. The 'species per path' approach [9] is related in the competing sub-population regard but only took advantage of a single transformation in creating the paths and associated and subpopulations. Further work with using the co-testability transformation approach to trap floating-point errors will seek to improve the fitness landscape so that the power of meta-heuristic search can be properly exploited.

Acknowledgements. The author would like to thank Mark Harman and Kiran Lakhota for useful conversations regarding this work.

References

- [1] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 21(1):5-48, 1991.
- [2] M. Harman. Open problems in testability transformation. In *Proceedings of the 1st International Workshop on Search-Based Testing*. IEEE digital library.
- [3] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation - program transformation to improve testability. In *Formal Methods and Testing, Lecture Notes in Computer Science*, volume 4949, pages 320-344. Springer-Verlag, 2008.
- [4] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3-16, 2004.
- [5] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. *ISSTA 2007*, pages 73-83, London, UK, 2007. ACM Press.
- [6] J. R. Hubbard. *Programming with C++*. Schaum's Outlines, 2nd ed., 2000.
- [7] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105-156, 2004.
- [8] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, To appear.
- [9] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. *ISSTA 2006*, pages 13-24, Portland, Maine, USA, 2006. ACM.
- [10] J. Zukowski. The need for BigDecimal, http://blogs.sun.com/CoreJavaTechTips/entry/the_need_for_bigdecimal