2008 International Conference on Software Testing, Verification, and Validation

# A Fitness Function to Find Feasible Sequences of Method Calls for Evolutionary Testing of Object-Oriented Programs

**(An Extended Abstract)**

Myoung Yee Kim and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, TX 79968-0518
{mkim2, ycheon}@utep.edu

## Abstract

*In evolutionary testing of an object-oriented program, the search objective is to find a sequence of method calls that can successfully produce a test object of an interesting state. This is challenging because not all call sequences are feasible; each call of a sequence has to meet the assumption of the called method. The effectiveness of an evolutionary testing thus depends in part on the quality of the so-called* fitness function *that determines the degree of the fitness of a candidate solution. In this paper, we propose a new fitness function based on assertions such as method preconditions to find feasible sequences of method calls. We show through experiments that to obtain the best search result the fitness function should consider the structures of method call sequences, which are essentially trees of assertions. We also provide a framework for combining multiple fitness values and for analyzing different fitness functions.*

## 1 Introduction

In unit testing object-oriented programs, test objects are constructed indirectly as sequences of method or constructor calls. However, it is difficult to find a call sequence that can successfully produce an object of an interesting state because not all call sequences are feasible. A call sequence is *feasible* if each call of the sequence, including one from those for creating argument objects, terminates normally without throwing an exception. In general, each call of the sequence has to meet the assumption of the called method, which is often formally written as a runtime checkable assertion such as a method precondition.

Meta-heuristic information can be used to guide the search for feasible call sequences [6]. In genetic algorithms, for example, call sequences that are likely to become feasible are selected and then made to evolve by applying genetic operations such as mutation and crossover. Thus, for an evolution approach to be effective, it is crucial to identify call sequences that have a potential for quickly becoming feasible. This is the responsibility of the so-called *fitness function* or *objective function* that measures the goodness of candidate solutions.

In this paper, we propose a new fitness function for evolutionary testing of object-oriented programs. Our fitness function is based on assertions such as method preconditions and thus views a call sequence as a tree of assertions to satisfy. It combines multiple fitness values given by the assertions of the tree by assigning them different weights. Because of dependencies among method calls, an assertion or some parts of it may not be evaluated, thereby producing no or a partial fitness value. Our fitness function can handle this kind of undefinedness in assertions by assigning a penalty for the undefined assertion or term.

We performed several experiments with our fitness function. The main finding from the experiments is that for the best result the fitness function should consider the structures of call sequences. In general, it suffices to assign weights to assertions based on levels of the assertions in the tree, though the optimal weight distribution depends on the complexities of and dependencies among the assertions. Assigning penalties to undefined assertions also improves the effectiveness of a fitness function.

## 2 Formulation of the Problem

The specific problem of our interest is: *how to define an effective fitness function based on method preconditions to guide the search for feasible call sequences?* The fitness

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.31

```
1  public class Account {
2    private /*@ spec_public @*/ int bal;
3
4    /*@ requires amt >= 0;
5      @ assignable bal;
6      @ ensures bal == amt; @*/
7    public Account(int amt) { bal = amt; }
8
9    /*@ requires amt > 0 && amt <= acc.bal;
10     @ assignable bal, acc.bal;
11     @ ensures bal == \old(bal) + amt
12     @   && acc.bal == \old(acc.bal - amt); @*/
13   public void transfer(int amt, Account acc) {
14     acc.withdraw(amt); deposit(amt);
15   }
16
17   /*@ requires amt > 0 && amt <= bal;
18     @ assignable bal;
19     @ ensures bal == \old(bal) - amt; @*/
20   public void withdraw(int amt) {
21     bal -= amt;
22   }
23
24   // The rest of definition ...
25 }
```

**Figure 1. Example JML specification**

function takes a call sequence as the input and returns a fitness value as the output. We assume that the class under consideration is annotated with a runtime-checkable specification; in particular, each method is annotated with a precondition. We are not much concerned with calculating a fitness value for a single method call; for this, we adopt an existing approach such as our earlier work [2]. Instead, we focus on combining multiple fitness values given by multiple calls of a call sequence by considering the structure of the call sequence.

As we use only the preconditions of method calls in the fitness calculation, we can abstract call sequences to tree structures called *assertion trees*, where a leaf node represents a primitive value or a no-argument constructor call, and a non-leaf node represents an object as a method or constructor calls with the receiver and arguments as its children. A node representing a method or constructor call is associated with an assertion, the precondition that the call has to satisfy.

As an example, let us consider an Account class shown in Figure 1; its behavior is formally specified in JML, a formal behavioral interface specification language for Java [5]. The following is an example call sequence for the Account class of which the assertion tree is shown in Figure 2.

[Account(10); withdraw(10);
transfer(20, [Account(0); withdraw(10)])]

As shown above, a call sequence may have subsequences because the arguments of a method or constructor call may be objects, represented as call sequences.

In summary, we are interested in defining a high quality fitness function—one that can identify and distinguish a

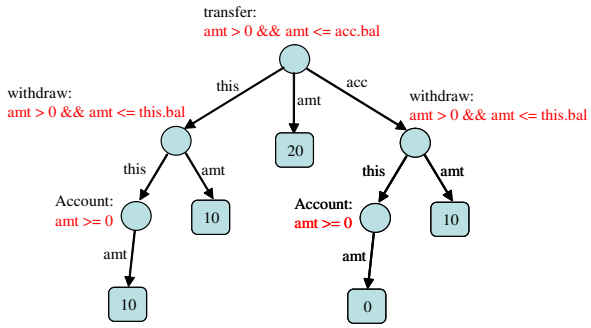



**Figure 2. Example assertion tree**

promising call sequence, thereby expediting the search for a feasible call sequence. By the quality or performance of a fitness function we mean the search speed measured in terms of the number of generations needed to find a solution, rather than the computational time spent for the search or the fitness calculation. In essence, the fitness function takes an assertion tree as the input and produces a fitness value for the tree by combining the fitness values given by the assertions of the tree.

## 3 Issues and Approaches

There are several issues that need be addressed on calculating fitness values for assertion trees, and each of these issues becomes an independent design dimension in defining a fitness function. The three issues that we consider in this paper are: (1) Which nodes—thus assertions—contribute to the calculation of the fitness value of the tree? (2) How much does each node contribute to the final fitness value of the tree? (3) How to handle undefinedness of the whole or parts of an assertion caused by dependencies among assertions. These issues or design dimensions can also be used as a framework for analyzing an existing fitness function.

Ideally we'd like to consider all the nodes of the tree when calculating the fitness value. However, because of time and other constraints, it may be advantageous to take into account only a subset of nodes; this often makes sense because the fitness value is an approximate value anyway. We can think of various techniques for choosing a subset of nodes, e.g., random sampling and selecting nodes based on certain conditions.

We conjecture that the structures of assertion trees and relative complexities of the assertions may influence the evolutions of call sequences. There is also a dependency between parent and child nodes of assertion trees. A parent object can be constructed only if all its child objects can be constructed. Thus, a parent's assertion may not be evaluated nor satisfied if the assertions of its children are not satisfied. Therefore, we believe that by varying each assertion's con-

tribution to the final fitness value we can produce a better fitness value. To control the influence of each assertion to the final fitness value, we introduce fitness weights.

One unique challenge of calculating fitness values for call sequences is that some of the assertions involved may not be evaluated for fitness calculation, thereby causing their values being undefined. For example, consider the call sequence of Section 2 of which the assertion tree is shown in Figure 2. The second conjunct of the `transfer` method's precondition, `amt <= acc.bal`, can't be evaluated because the call sequence for the argument `acc` is infeasible. The requested withdrawal amount is larger than the available balance. As the object `acc` can't be built, the term `acc.bal` can't be evaluated and thus is undefined. Our approach to coping with this kind of undefinedness in assertions is to identify the smallest boolean expression that encloses an undefined term (e.g., `amt <= acc.bal`) and assign a penalty as its fitness value. This approach is called a contextual interpretation of undefinedness [3].

# 4 Algorithm

Our fitness function is generic in that it is parameterized with the node weight as well as the penalty value for undefinedness. Thus, by varying the parameter values, we can produce a family of fitness functions (see Section 5). The fitness calculation is done in two steps by first distributing the weight to each node of the assertion tree and then calculating the fitness of each node and summing them by recursively traversing the tree.

The node weight can be distributed evenly to each node of the tree. That is, the weight of a node $i$, $W_i$, is defined as $W_i = 1/N$, where $N$ is the number of the nodes in the tree. Alternatively, it can be distributed based on the depth of the node in the tree, e.g., to give more weights to child nodes than parent nodes. In this depth-based weight distribution, the fitness function takes the weight ratio between the child and parent nodes as a parameter and calculates the weight for each node, as follows.

$$W_i = (1/\sum_{j=1}^{N} R^{D_j-1}) * R^{D_i-1}$$

where $N$ is the number of nodes in the tree, $R$ is the weight ratio, and $D_i$ is the depth of a node $i$. If $R$ is 10, for example, each child node has 10 times more weight than its parent node.

To handle undefinedness, the fitness function is parameterized with a penalty value and a flag indicating the use of contextual interpretation. If the flag is true, the contextual interpretation is employed that identifies the smallest boolean expression enclosing an undefined term; otherwise, the whole assertion is interpreted as undefined. The penalty value is used in place of an undefined expression.

# 5 Experiment

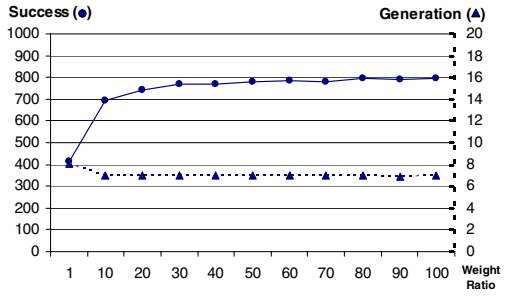
We performed several experiments to study the effects of fitness parameters such as the weight distribution and the penalty value on the effectiveness of fitness functions. We also compared a number of sample fitness functions. For our experiments, we chose the Account class introduced in Section 2. The goal was to find a feasible call sequence that can successfully construct an Account object. However, to simplify our experiments, as well as to minimize the influence of other aspects of evolutionary testing, we fixed the structure of the call sequence to that of the sample sequence shown in Section 2. We only allowed the evolution of integer argument values—leaf nodes in the assertion tree. Thus, the search goal was to find five integer values that satisfy all the assertions of the assertion tree (see Figure 2).

We fixed the population size of each generation to 10 chromosomes, each chromosome consisting of five integer values corresponding to the leaf nodes of the assertion tree. We populated the initial generation randomly and used a rank-based selection to determine candidates for the subsequent generation. For evolution, we used only a mutation operator that increments or decrements the genes of a chromosome by a small, randomly-chosen number. The evolution was repeated until a solution was found or a maximum number of generations (100) was reached.

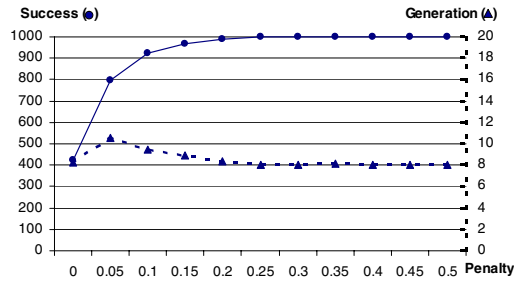
Figure 3 shows the results of our experiment. For each value of the control parameter (e.g., weight ratio), we ran our program 1000 times and recorded the number of times a solution was found and the average number of generations needed to find a solution. As shown in Figure 3(a), a fitness function becomes more effective with more weights on child nodes; for this particular experiment, we fixed the penalty value to 0 and used no contextual interpretation, but it was true in general for other values. Figure 3(b) shows a similar result for penalty values; i.e., heavier penalties also improve the effectiveness of fitness functions. It should be noted, however, that by increasing the penalty we achieved a 100% success rate while by varying the weight we were only able to achieve a 80% success rate. Figure 3(c) shows the effectiveness of several sample fitness functions with different parameter values. A general conclusion is that the weight, the penalty, and the contextual interpretation in isolation all have positive effects on the effectiveness of fitness functions. However, with a large enough penalty value the other two parameters have no effect.

# 6 Related Work

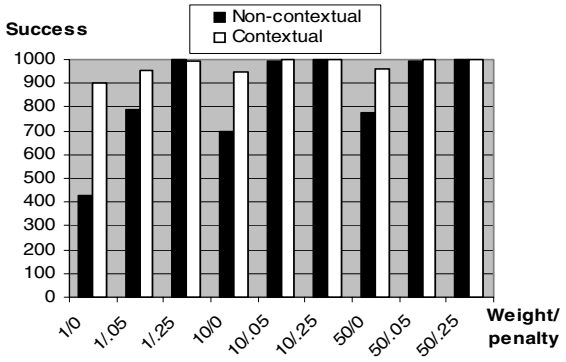
Several researchers used weights and penalty values in the applications of genetic algorithms. For example, Harman et al. proposed a multi-objective approach for searching test cases that satisfy several goals at the same time,




(a) Effect of weight




(b) Effect of penalty




(c) Sample combinations of weight and penalty

**Figure 3. Experimental results**

e.g., branch coverage and dynamic memory allocation [4]. They assigned a different weight to each fitness function to control the contribution that each function makes to the overall fitness value. In our approach, the weight is calculated based on the structure of a candidate solution and controls the contribution of each part of the candidate to the final fitness value. Burke and Newall applied a multistage evolutionary algorithm to a timetable scheduling problem [1]. A candidate solution can have one extra time period where unscheduled events can be placed, and such a candidate receives a heavy penalty. They reported that, to a certain threshold value, a heavy penalty value improved the performance of the fitness function, which is similar to our own findings from our experiments.

Our approach for handling undefinedness in assertions was inspired by the work of Cheon and Leavens [3], where an occurrence of undefinedness is interpreted as either true or false depending on the context of its occurrence. We adapted the approach for the fitness calculation in that we assign a penalty value $p$ $(0 \leq p \leq 1)$ or $1-p$ to the smallest boolean expression enclosing the undefined term.

## 7 Conclusion

We developed a fitness function for evolutionary testing of object-oriented programs. The search goal for the fitness function is to find a sequence of method calls that builds a test object successfully by satisfying the method precondition of each call in the sequence. Thus, our fitness function is defined in terms of the preconditions of the method call sequence. We used weights to control the contribution that each call in the sequence makes to the fitness value, and penalty values to handle undefinedness in assertions caused by dependencies among the assertions. Our experiments showed that both weights and penalty have positive effects on the effectiveness of a fitness function, measured in terms of the success rate and the number of generations needed.

## Acknowledgment

This work was supported in part by NSF under Grant No. CNS-0509299 and CNS-0707874. Thanks to anonymous referees for comments.

## References

[1] E. K. Burke and J. P. Newall. A multistage evolutionary algorithm for timetable problem. *IEEE Transactions on Evolutionary Computation*, 3(1):63–74, Apr. 1999.

[2] Y. Cheon and M. Kim. A fitness function for evolutionary testing of object-oriented programs. In *Genetic and Evolutionary Computation Conference, Seattle, WA, USA, July 8-12, 2006*, pages 1952–1954. ACM Press, July 2006.

[3] Y. Cheon and G. T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 19–21, 2005*, pages 149–157. ACM Press, Sept. 2005.

[4] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Genetic and Evolutionary Computation Conference, London, England, United Kingdom*, pages 1098–1105, July 2007.

[5] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[6] P. Tonella. Evolutionary testing of classes. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA*, pages 119–128, July 2004.