A Specification-Based Fitness Function for Evolutionary Testing of Object-Oriented Programs*

Yoonsik Cheon and Myoung Kim
University of Texas at El Paso
Department of Computer Science
El Paso, Texas 79968
{ycheon, mkim2}@utep.edu

ABSTRACT

Encapsulation of states in object-oriented programs hinders the search for test data using evolutionary testing. As client code is oblivious to the internal state of a server object, no guidance is available to test the client code using evolutionary testing; i.e., it is difficult to determine the fitness or goodness of test data, as it may depend on the hidden internal state. Nevertheless, evolutionary testing is a promising new approach of which effectiveness has been shown by several researchers. We propose a specification-based fitness function for evolutionary testing of object-oriented programs. Our approach is modular in that fitness value calculation doesn't depend on source code of server classes, thus it works even if the server implementation is changed or no code is available—-which is frequently the case for reusable object-oriented class libraries and frameworks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; I.2.8 [**Artificial Intelligence**]: Program Solving, Control Methods, and Search—*heuristic methods*

General Terms

Verification

Keywords

evolutionary testing, fitness function, formal specification

1. INTRODUCTION

In evolutionary testing, one uses metaheuristic search techniques such as genetic algorithms to select or generate test data [4]. The search space is the input domain of the program under test, and the problem is to find a (minimal) set of input data, called *test cases*, that satisfies a certain test criterion, such as branch coverage and condition coverage. Unlike an analytical algorithm that solves a set of constraints specifying the test criterion, an evolutionary algorithm uses simulated evolution as a search strategy to evolve candidate solutions, i.e., test cases, using operators inspired by genetics and natural selection. A key component of evolutionary algorithms is a *fitness* or *objective function* that measures the goodness of candidate solutions [1]. A candidate survives and thus evolves into a new generation based on its fitness value obtained by applying the fitness function. Therefore, for an evolutionary approach to be effective, the fitness function should be able to identify promising candidates—fitter ones—from those that are not so promising; otherwise, the search becomes a random search. Several researchers applied evolutionary algorithms to testing and showed their effectiveness (c.f. [4, 6]).

In object-oriented programs, the state of an object is hidden and is accessible only through a set of exported or public methods, called an *interface*. Encapsulating object states is an excellent tool for modularizing programs, as changes to implementation decisions and details such as data structures and algorithms don't affect client code as long as the interface remains the same. However, encapsulation becomes problematic when testing object oriented programs [3]. It is hard or sometimes impossible to create a test object with a desired state, as one cannot directly manipulate the hidden state variables, and similarly it is difficult or impossible to observe the effect of method execution because one cannot directly access the state variables. The problem becomes aggravated when testing object-oriented programs using evolutionary techniques. The hidden state prevents one from measuring accurately the fitness of an individual candidate object. Without an accurate fitness measurement, no guidance would be provided to search for better test data.

We propose a new fitness function for evolutionary testing of object-oriented programs. The key idea of our approach is to use the behavioral specification of a method to determine the goodness or fitness of test data.

2. THE PROBLEM

Suppose that the current search goal is to find a `Course` object, `c`, that satisfies the condition `c.isClosed()`, where `isClosed` is a boolean method defined for the class `Course` and returns true if a course is closed, e.g., no seat is available. If all candidate solutions fail the condition, which ones should be chosen for further evolution? As the course state is hidden and the `isClosed` method returns false for all candidates, it is impossible to make a sensible choice. As a result, there is little guidance to an evolutionary search in selecting candidate objects. This is due to the boolean method's "all or nothing" nature [5]. All candidate objects failed the

*A full version of this paper is available as a technical report, TR-05-35, at `www.cs.utep.edu`

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
ACM 1-59593-186-4/06/0007.

condition, and no one failed better than the others; i.e., we cannot differentiate them. However, our intuition tells us to choose the course object with the smallest number of seats available.

3. OUR APPROACH

Our approach is, for a method call, to use the method's specification to calculate the fitness value of the method call expression. For example, suppose that the specification of the `isClosed` method be written in JML [2], a behavioral interface specification language for Java, as follows.

```
//@ ensures \result <==> (size >= maxSize);
public boolean isClosed() { /* ... */ }
```

The postcondition written in the `ensures` clause states that the return value, denoted by `\result`, is true if and only if `size` is greater than or equal to `maxSize`; private fields `size` and `maxSize` represent the number of current enrollments and the maximum number of allowed enrollments, respectively. In our approach, we substitute `isClosed()` with the expression `size >= maxSize` when calculating the fitness of each candidate; i.e., the fitness function f becomes:

$$\begin{aligned} & f(\texttt{isClosed()}) \\ & \quad = f(\texttt{size >= maxSize}) \\ & \quad = \begin{cases} 0 & \text{if } \texttt{maxSize - size} \leq 0 \\ \texttt{maxSize} - \texttt{size} & \text{otherwise} \end{cases} \end{aligned}$$

Thus, our approach can differentiate among failed candidates by assigning different fitness values, thus providing a better guidance toward the search goal. Indeed, for this particular case, the guidance is accurate and matches our intuition by letting us to choose the best candidate available.

More formally, given a boolean method m, we define the fitness function for a method call, $e_0.m(e_1, \ldots, e_n)$, as:

$$f(e_0.m(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} f(Spec_m^T[e_0, e_1, \ldots, e_n])$$

where T is the static or dynamic type of e_0, $Spec_m^T$ denotes the specification of the method m found in the type T, and $Spec_i^T[e_0, e_1, \ldots, e_n]$ means to substitute $e_0, e_1, \ldots, e_n$ for formal parameters, including `this`, to evaluate the specification with the receiver e_0 and actual arguments $e_1, \ldots, e_n$. The specification to be evaluated can be determined statically at compile time based on the static type of the receiver, e_0, or dynamically at runtime based on the runtime type of e_0. The former is easier to implement while the latter gives a more accurate fitness value. Even if the method is overridden in a subclass, the static approach still gives a meaningful measurement because a subtype has to preserve its supertype's specification.

4. PRELIMINARY RESULTS

A preliminary experiment indicates that our fitness function improves an evolutionary search greatly from 300% up to 800% in terms of the number of iterations needed to find a solution. The improvement varies depending on the various parameters of the evolutionary approach. Other advantages of our approach is that it doesn't depend on the source code of called methods, it doesn't require a whole program analysis, and it works even in the presence of method overriding and dynamic binding. In general, source code-based techniques don't work for object-oriented programs because, due to method overriding and dynamic binding, the actual method to be invoked and thus the source code to use in calculating fitness values cannot be determined statically. Worse, the source code of called methods may not be available. This is frequently the case for reusable object-oriented class libraries and frameworks, as software vendors are reluctant to distribute their source code.

5. CONCLUSION

The effectiveness of evolutionary testing is determined in part by its fitness function that provides a guidance to the search by telling how good each candidate solution is. The hidden state is a serious barrier to applying evolutionary testing to object-oriented programs. Our solution to this problem is to use a boolean method's specification to calculate fitness values. The solution is evolutionary in that it can be incorporated into existing fitness functions or testing methods. A preliminary result shows that our specification-based fitness function outperforms the fitness functions that don't use the specification. We believe that as the goal condition (or predicate) becomes more complex, our fitness function becomes more effective.

The next step of our research is to show the practicality of our approach. Toward this end, we are building an automated, evolutionary testing tool for Java by integrating JML and JUnit. Our plan is to apply the new fitness function to this tool and evaluate its practicality. Another direction of future research is to explore the use of JML specifications to extract meta information about the relationships between state variables. Such meta information may be useful not only for computing fitness functions but also for improving evolutionary testing in general, e.g., performance.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation, under Grant No. CNS-0509299.

7. REFERENCES

[1] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002, New York, July 9-13, 2002*, pages 1329–1336.

[2] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[3] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

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

[5] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *GECCO 2003, Chicago, July 12-16, 2003*, volume 2724 of *Lecture Notes in Computer Science*, pages 2488–2500. Springer-Verlag, 2003.

[6] P. Tonella. Evolutionary testing of classes. In *ACM SIGSOFT ISSTA 2004, Boston, MA*, pages 119–128, July 2004.