10-2005

A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs

Yoonsik Cheon
The University of Texas at El Paso, ycheon@utep.edu

Kim Myoung
The University of Texas at El Paso, mkim2@utep.edu

Comments:
UTEP-CS-05-35.

A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs

Yoonsik Cheon and Myoung Kim

TR #05-35
October 2005; revised January 2006

Keywords: fitness function, evolutionary testing, genetic algorithms, branch coverage, test data generator, pre and postconditions, JML language.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — class invariants, formal methods, programming by contract; D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and postconditions, specification techniques; I.2.8 [*Artificial Intelligence*] Problem Solving, Control Methods, and Search — Heuristic methods.

This is an extended version of the paper appeared in *Genetic and Evolutionary Computation Conference, Seattle, WA, USA, July 8-12, 2006, pages 1952-1954, ACM Press, 2006.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs

Yoonsik Cheon*and Myoung Kim
Computer Science
The University of Texas at El Paso
500 W. University Avenue
El Paso, Texas 79968-0518
cheon@cs.utep.edu

Abstract

We show that encapsulation of states in object-oriented programs hinders the search for test data using evolutionary testing. In a well-designed object-oriented program the encapsulated or hidden state is accessible only through exported or public methods. 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. In particular, it is difficult to determine the fitness or goodness of test data, as it may depend on the hidden internal state. However, evolutionary testing is a promising new approach whose 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 still 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. Our approach works for both black-box and white-box based testing.

1 Introduction

In evolutionary testing one uses metaheuristic search techniques, such as genetic algorithms [7], to select or generate test data [12]. 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 satisfy 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 (c.f. [5]), 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 measure the goodness of candidate solutions. A candidate survives and thus evolves into a new generation based on its fitness value obtained by applying the fitness function. 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 [1, 12, 14, 16, 20], though most considered only procedural programs with simple data types such as integer.

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 and increasing reusability of 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. 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

*The work of the author was supported in part by NSF under the grant number CNS-0509299.

one cannot directly access the state variables [10].

The problem becomes serious when testing object-oriented programs by using evolutionary techniques. It is difficult to measure accurately the fitness of an individual candidate object because its state may be hidden and may not be directly accessible for calculating the fitness value. That is, it often becomes impossible to assess the goodness of test data. Without accurate fitness measurement, no guidance is provided in searching for better or improved test data. For example, suppose that the search goal is to find a list `l` that satisfies the condition `l.isEmpty()`, where `isEmpty` is a boolean method defined for lists. If no solution is found, i.e., all candidates fail the condition, which ones do we choose for further evolution? As the list state is hidden and the `isEmpty` method returns false, it is impossible to make a sensible choice in selecting candidates for further evolution among the failed ones. As a result, there is little guidance to the evolutionary search in selecting candidate objects. This is due to the boolean method's "all or nothing" nature. All candidate objects failed the condition, and no one failed better than the others; i.e., we cannot differentiate them. However, our intuition says to choose the one with the shortest length.

In this paper we propose a new approach to defining fitness functions for evolutionary testing that allows one to use such an intuition as "choosing the one with the shortest length." The core idea of our approach is to use the behavioral specification of a method to determine the goodness (or fitness) of test data. For example, we use the `isEmpty` method's specification in calculating fitness values. If the specification is written in terms of or related to the length of lists, lists of different lengths will receive different fitness values.

A preliminary experiment shows that our approach improves the search 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. Another strength of our approach is that it is modular in that it doesn't depend on the source code of called methods. In general, source code-based techniques to calculating fitness values don't work for object-oriented programs because, due to method overriding and dynamic dispatch, the actual method to be invoked and thus the source code to use in calculating fitness values cannot be determined statically. In addition, the source code of called methods may not be available for testing. This is frequently the case for reusable object-oriented class libraries and frameworks, as software vendors are reluctant to ship their source code.

The rest of this paper is organized as follows. In Section 2 we explain evolutionary testing, focusing on the roles of fitness functions. In Section 3 we describe the hidden state problem of object-oriented programs for evolutionary testing through an example; the hidden state of an object makes it difficult to measure the fitness of candidate test cases accurately. In Section 4 we explain our approach in detail by defining a new specification-based fitness function, and preliminary experimental results of our approach are summarized in Section 5. In Section 6 we discuss related work, and in Section 7 we conclude this paper with a summary.

2 Evolutionary Testing

In evolutionary testing, metaheuristic search techniques such as genetic algorithms are used to select or generate test data. The search space is the input domain of the program under test. The search starts with an initial set of test cases that are typically generated randomly, and then evolves them into new generations by applying operations inspired by genetics and natural selection, such as selection, crossing, and mutation. The process is repeated until a solution—a set of test cases that satisfy the testing criterion—is found or a certain stopping condition, e.g., the maximum number of iterations, is met. The search is guided by a fitness function that calculates the fitness values of the individuals in the generation in that the fitter ones have a higher chance to survive and thus evolve into the next generation. Thus, the effectiveness of an evolutionary testing is determined in part by the quality of the fitness function.

An evolutionary approach can be applied to both black-box and white-box testing, though in this paper we explain our approach only by using white-box testing. In white-box testing, the basis of testing is the structure of the program under test. For example, one common test coverage criterion, called *branch coverage*, is to execute all branches of the program's control flow graph. In this case, the search goal becomes to find a set of test cases that execute every branch of the program under test.

Wegener et al. claims that a higher level of coverage can be obtained when each branch is targeted individually as a partial aim [20]. They use the *branch*

distance to indicates how close an individual test case was to evaluating the target branch condition in the desired way. For example, if the target condition is $x == y$, the branch distance is calculated by $|x - y|$. The branch distance is one element of calculating the fitness values of individuals in the population. The other element, called the *approximation level*, indicates how close an individual is to reaching the target branch condition; there may be other branch conditions between the start statement and the target branch statement that must be satisfied to reach the target condition. However, as the other elements such as the approximation level are orthogonal to our approach, we will only consider the branch distance when we define fitness functions in this paper.

3 The Hidden State Problem

In this section we explain the hidden state problem of applying evolutionary testing to object-oriented programs.

The class `Course` in Figure 1 describes courses offered by a university. The partial code shown addresses only the enrollment aspect of the courses. Each course has a fixed, maximum number of students allowed to be enrolled (`maxSize`), and a student can enroll for a course as long as the course is not closed. A course becomes closed when the number of enrollments (`size`) reaches the maximum enrollment number (`maxSize`).

Suppose we are interested in testing the `enroll` method of the class `Course`. A test case in this case consists of two objects, a course object and a student object, because we have to send an enroll message to some course object, i.e., receiver, to test the `enroll` method. Suppose we do white-box testing and our test objective is to cover all branches. This test criterion is called *branch coverage*, and its objective is to find a (minimal) set of test cases that execute all branches of the code under test. If the current goal is to cover the true branch of the `if` statement, we need to find a test case that satisfies the condition `isClosed()`. Suppose we have in the current population a set of test cases, $\langle c_i, s_i \rangle$ $(i = 1, \ldots, n)$, where c_i is an instance of class `Course` and s_i is a `Student` object. If any individual of the population makes the condition to hold, i.e., c_i`.isClosed()`, we found a solution and thus stop the search. Otherwise, we select some number of individuals from the current population to let them evolve into the next generation population and continue our search. Of course, we would like to select those individuals that are most promising so that the search progresses toward a solution. Determining whether an individual is promising or not is the role of a *fitness function*. A fitness function gives a measurement that tells how close an individual of a population is to the solution being searched. That is, it determines the goodness of each individual of the population to guide the search toward a solution.

```
public class Course {
  private final int maxSize;

  private int size;

  public Course(int maxSize) {
    this.maxSize = maxSize;
  }

  public boolean isClosed() {
    return size >= maxSize;
  }

  public boolean enroll(Student s) {
    if (isClosed()) {
       return false;
    } else {
       // add s to this course ...
       return true;
    }
  }

  // other fields and methods
}
```

Figure 1: Partial definition of class `Course`

The problem here is that the conventional approach to calculating fitness values from the goal expression doesn't give any guidance to search. Each test case $\langle c_i, s_i \rangle$ will receive the same fitness value, as c_i`.isClosed()` becomes false. The fitness function fails to return better values for those test cases that almost satisfy the goal expression, and worse values for those that are a long way from satisfying the expression. As a result, an evolutionary search becomes down-graded into a random search. This is due to the "all or nothing" nature of boolean methods and the encapsulation of the state [11]. An individual makes the boolean method either true (in which case a solu-

tion is found) or false (in which case the individual is not a solution, but the method doesn't tell how badly the individual failed to satisfy the method). The state of an object, such as `maxSize` and `size`, is hidden or encapsulated inside the object, and clients are required to use well-defined interface methods such as `isClosed()` to observe the state of the object. If the condition had been written directly in terms of state variables, e.g., `size >= maxSize`, the evolutionary search would be better guided because one can use, for example, the difference between `size` and `maxSize` as a fitness value. Note that such a rewriting is not always possible, as the called method may be defined in another class and written in terms of some private fields.

In sum, the use of boolean method such as `isClosed` produces the same fitness value for all non-solution individuals, and thus fitness values provide little guidance to the evolutionary search for identifying input values for the true case in the event that current test data had led to the false case, or vice versa. However, fitness values play a key role in the evolutionary search, as it guides the search; if all fitness values becomes the same, the search basically becomes a random search.

In the next section we define a new fitness function that can produce good fitness values in the presence of boolean methods in the goal expression.

```
public class Course {
  private /*@ spec_public @*/
    final int maxSize;
  //@ public invariant maxSize > 0;

  private /*@ spec_public @*/ int size;
  //@ public invariant
  //@   0 <= size && size <= maxSize;

  //@ requires maxSize > 0;
  public Course(int maxSize) {
    this.maxSize = maxSize;
  }

  //@ ensures \result == (size >= maxSize);
  public boolean isClosed() {
    return size >= maxSize;
  }

  public boolean enroll(Student s) {
    if (isClosed()) {
       return false;
    } else {
       // add s to this course ...
       return true;
    }
  }
}
```

Figure 2: Specification of class `Course`

4 Our Approach

We first explain our approach through an example and then rigorously define a new fitness function that is the core of our approach.

The underlying idea of our approach is simple. As in the earlier work, we calculate the branch distance based on the structure of a goal expression. For a method call, however, if the called method returns a boolean value, the method's specification is used to calculate the fitness value of the method call expression. In short, the behavioral specification of a method provides a guidance toward the goal.

Figure 2 shows a specification of class `Course` written in the Java Modeling Language (JML). JML is a behavioral interface specification language for Java and allows to formally describe the behavior of Java modules such as classes and interfaces [9]. In JML, specifications are typically annotated in source code as a special kind of comments such as `//@` and `/*@ @*/`. The behavior of a module is described by writing invariants in the `invariant` clause and method pre and postconditions in the `requires` and `ensures` clauses, respectively.

The two invariants of Figure 2 state that `maxSize` is always positive and `size` is always between 0 and `maxSize`, inclusive[1]. The postcondition of the `isClosed` method states that the return value, denoted by `\result` is true if and only if `size` is greater than or equal to `maxSize`.

As in the previous section, let us assume the current goal is to cover the true branch of the `if` statement of the `enroll` method. That is, the goal is to satisfy the condition `isClosed()`. The difference in our approach is that we substitute a method call with its behavioral specification when calculating a

[1]The JML modifier `spec_public` states that private fields such as `maxiSize` and `size` may be used in public specifications, e.g., public invariants and pre and postconditions of public constructors and methods, such as `Course` and `isClosed`.

fitness value of a test case. In this case, we substitute `isClosed()` with the expression `size >= maxSize`, and the fitness function f is defined as follows.

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

Here smaller values denote fitter ones; later in Section 4.1, we will give a slightly different definition when we formally define the fitness function.

Now let us consider several sample test cases. A test case for the `enroll` method consists of a `Course` object (the receiver) and a `Student` object (the argument). As the argument doesn't contribute to the calculation of fitness values, we won't show them. The following table shows four different test cases and their fitness values. A `Course` object is represented as a tuple of `maxSize` and `size`.

Id	[maxSize, size]	Fitness
T1	[10, 1]	9
T2	[10, 3]	7
T3	[10, 6]	4
T4	[10, 9]	1

Test case T4 is the fittest one. This makes a sense, as it requires one additional `enroll` call to satisfy the goal condition `isClosed()` while others requires at least 4, 7, and 9 `enroll` calls, respectively. Note that all the above test cases make the condition `isClosed()` false, thus the conventional approach cannot differentiate them in selecting best candidates for further evolution. Our approach, on the other hand, is able to assign different fitness values to them, therefore, 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.

In the following subsection we define the fitness function formally.

4.1 A New Fitness Function

In this section we explain our approach by defining a fitness function for a subset of JML expressions. These are expressions used to write method specifications and include Java expression. We first define the fitness function for Java expressions and later extend it to include JML-specific expressions.

$$
\begin{aligned}
&I \in \text{Identifier} \\
&E \in \text{Expression} \\
&E ::= I \\
&\quad | \; E_1\texttt{.}I \\
&\quad | \; E_0\texttt{.}I\texttt{(}E_1\texttt{, ..., }E_n\texttt{)} \\
&\quad | \; \texttt{!}E_1 \\
&\quad | \; E_1 \texttt{ || } E_2 \\
&\quad | \; E_1 \texttt{ \&\& } E_2 \\
&\quad | \; E_2 \texttt{ == } E_2 \\
&\quad | \; E_1 \texttt{ != } E_2 \\
&\quad | \; E_1 \diamond E_2
\end{aligned}
$$

Figure 3: Abstract syntax of a subset of Java expressions. The meta symbol $\diamond$ stands for relational operators such as `<`, `<=`, `>`, and `>=`.

Table 1: Fitness function for Java boolean expressions.

Expression	Fitness function f
v	some constant c
$e\texttt{.}v$	$f(v)$
$\texttt{!}e$	$1 - f(e)$
$e_1 \texttt{ || } e_2$	$\max(f(e_1), f(e_2))$
$e_1 \texttt{ \&\& } e_2$	$\min(f(e_1), f(e_2))$
$e_1 \texttt{ == } e_2$	if e_1 is not numeric, some constant d
$e_1 \texttt{ != } e_2$	$1 - f(e_1 \texttt{ == } e_2)$

Figure 3 shows a subset of Java expressions that we will define a fitness function for. A fitness value is a real number in the range of 0 and 1, inclusive, and a higher number denotes a fitter one. The reason for this is that we want to use an expression's probability of being true as its fitness value.

Table 1 shows the definition of fitness function for some of Java boolean expressions. A reasonable value for the constant c, the fitness of a boolean variable, would be 0.5, and a reasonable value for the constant d would be a very small number, the probability that two expressions denote the same value or object. A nice consequence of using values between 0 and 1 as fitness values is that the fitness values of negation expressions, such as $\texttt{!}e$, are intuitively defined as $1 - f(e)$, where f is the fitness function.

Table 2 shows the fitness function for relational expressions. A similar definition is found in the literature of search-based test data generation [8, 18]. However, one difference is that our definition uses a

Table 2: Fitness function for relational expressions. The *normalization function*, n: $R \rightarrow [0,1]$, scales non-negative numbers to the values between 0 and 1, inclusive.

Expression	Fitness function f
e_1 `==` e_2	if $e_1 - e_2 = 0$ then 1 else $n(\lvert e_1 - e_2 \rvert)$
e_1 `<` e_2	if $e_1 - e_2 < 0$ then 1 else $n(e_1 - e_2)$
e_1 `<=` e_2	if $e_1 - e_2 \leq 0$ then 1 else $n(e_1 - e_2)$
e_1 `>` e_2	if $e_2 - e_1 < 0$ then 1 else $n(e_2 - e_1)$
e_1 `>=` e_2	if $e_2 - e_1 \leq 0$ then 1 else $n(e_2 - e_1)$

normalization function to make all fitness values be within the interval [0,1].

We next define the fitness function for method call expressions, $e_0.i(e_1, \ldots, e_n)$, which is the heart of our definition. The return type of method i is assumed to be boolean.

$$f(e_0.i(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} f(Spec_i^T[e0, e_1, \ldots, e_n])$$

where T is the (static or dynamic) type of e_0, $Spec_i^T$ denotes the specification of method i found in type T, and $Spec_i^T[e_0, e_1, \ldots, e_n]$ means to substitute in $Spec_i^T$ $e_0, e_1, \ldots, e_n$ for `this` and formal parameters, respectively. The specification is evaluated in the context of the receiver e_0.

As mentioned earlier, the key idea is to use the specification of a boolean method in place of its call, as the resulting boolean value doesn't give any useful measurement for calculating fitness values. A method's specification typically consists of a pair of pre and postconditions, in which case it is desugared to $P \Rightarrow Q$, where P and Q are pre and postconditions, respectively. 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. Note that, even if the method is overridden in a subclass, the static type-based approach still gives a meaningful measurement because a subtype has to preserve its supertype's specification.

Finally we define the fitness function for JML-specific expressions. Figure 4 shows a subset of JML expressions, and Table 3 shows the fitness function. In the fitness definition, MIN and MAX denote the minimum and maximum values, respectively, of the given expression when evaluated once for each possible value of the quantified variable (refer to [4] for an approach to evaluating quantified expressions).

$I \in$ Identifier
$T \in$ Type
$D \in$ Declaration
$E \in$ Expression
E ::= ...
| `\result`
| E_1 `==>` E_2
| E_1 `<==` E_2
| `(\forall` D`;` E_1`;` E_2`)`
| `(\exist` D`;` E_1`;` E_2`)`
D ::= T I

Figure 4: Subset of JML expressions.

Table 3: Fitness function for JML-specific expressions.

Expression	Fitness function f
`\result ==` e	$f(e)$
`\result !=` e	$f(!e)$
e `== \result`	$f(e)$
e `!= \result`	$f(!e)$
`\result`	constant c
e_1 `==>` e_2	$f($`!`e_1 `||` $e_2)$
e_1 `<==` e_2	$f(e_1$ `||` `!`$e_2)$
`(\forall` d`;` e_1`;` e_2`)`	MIN($f(e_1$ `==>` $e_2)$)
`(\exists` d`;` e_1`;` e_2`)`	MAX($f(e_1$ `&&` $e_2)$)

5 Preliminary Results

We did a simple preliminary experiment to evaluate the effectiveness of our approach before we start a full-blown implementation. We implemented a simple genetic algorithm to generate test cases for the `enroll` method of the class `Course` presented in Section 4. The genetic algorithm initializes the population with a random set of test cases, C, and evolves it iteratively until a solution is found. A test case is a solution if it satisfies the condition of the `if` statement. At each iteration, the fitness of each $c \in C$ is calculated to rank the population by fitness. Some fraction of the lower-fitness individuals are discarded. They are replaced by new test cases obtained by applying genetic operations to the remaining test cases.

Table 4: Average number of iterations

pop. size	number of iteration		
	without spec	with spec	ratio
10	87.15	10.29	8.47
20	37.54	7.41	5.07
30	25.98	5.57	4.66
40	21.20	4.73	4.48
50	16.41	4.28	3.83
60	11.81	3.55	3.33
70	10.43	3.43	3.04
80	9.48	3.09	3.07
90	9.06	2.71	3.34
100	7.40	2.67	2.77

In this experiment we only used mutation. To bring diversity to the population, some fraction of the new generation are generated randomly.

We ran the algorithm and measured the number of iterations needed to find a solution. The result varies depending on various parameters of genetic algorithms, such as population size, mutation rate, and generation gap. In all cases, however, the specification-based fitness function as defined in Section 4.1 outperforms a fitness function that doesn't use specifications. For example, Table 4 shows the result of one particular experimental run. The performance improvement is between 277% to 847%. It is worth mentioning that, depending on the way a random test case is generated, the conventional fitness function failed many times to find a solution. However, the specification-based fitness function was always able to find a solution. It is our prediction that as the goal condition (or predicate) becomes more complex, the specification-based fitness becomes more effective.

6 Related Work

The use of flag variables [2, 6] and internal state variables [11, 15] in procedural programs have been shown to have a similar problem. They inhibit the search for test data. All the approaches known to us analyze source code in one way or another. For example, a common approach for a flag variable is to find the last definition point in the path from the start statement to the target condition statement and to replace the flag variable with its definition. This source code-based approach doesn't work for object-oriented programs because, due to dynamic dispatch of method calls, it is in general impossible to determine statically the actual code to be executed. An interface method (of Java) poses another problem, as there is no definition (i.e., source code) associated with it. The approach is not modular in the sense that it potentially requires a whole program analysis.

The definition of our fitness function is inspired by Korel's objective function [8] that was used to search test data satisfying each branch predicate of Pascal programs, and also by the cost function of Tracy et al. that was used to generate test data by using simulated annealing [18]. In addition to using specifications for boolean method calls, one main difference of our approach is that our fitness values are normalized, and as a result the fitness values for negation expressions are intuitively defined. Definitions similar to ours are found in fuzzy logic where a statement may have a degree of truth between 0 and 1, and a definite conclusion (i.e., true or false) is derived from vague, ambiguous, imprecise, noisy, or missing information (c.f. [13]).

Tracy et al. also used specifications—pre and postcondition pairs—to calculate fitness values [18]. Their goal was to apply genetic algorithm to black-box testing and to drive test cases towards detecting faults, i.e., violations of specifications. For this, they transformed the specification predicate into disjunctive normal form and made each conjunct contribute to the final fitness value. However, they didn't use the specifications of boolean functions in calculating the fitness values for function calls.

There are two approaches known to us that applied genetic algorithms to generate test cases for object-oriented programs [17, 19]. Both approaches used branch coverage, and the fitness values are calculated based on branch predicates and source code analysis. Neither addressed the hidden state problem.

7 Conclusion

The effectiveness of evolutionary testing is determined in part by its fitness function. A fitness function provides a guidance to the search by telling how good each candidate solution is. We showed that hidden state is a serious barrier to evolutionary testing of object-oriented programs. Because the hidden state can be observed only through observer methods, it is difficult to measure accurately the fitness of an

object. This is particularly true when the observer method is a boolean method. Our solution to this problem is to use a boolean method's specification to calculate fitness values. A preliminary experimental result shows that our specification-based fitness function outperforms the fitness function that doesn't use the specification up to 847% in terms of the number of iterations needed to find a solution. Our approach is modular; it doesn't require 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 dispatch.

The next step of our research is to show the practicality of our approach. We are in the process of building an automated, evolutionary testing tool for Java by integrating JML and JUnit [3]. Our plan is to apply the new fitness function to this testing tool and analyze its practicality.

References

[1] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[2] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[3] Yoonsik Cheon, Myoung Kim, and Ashaveena Perumendla. A complete automation of unit testing for Java programs. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume I, Las Vegas, Nevada, June 27-29, 2005*, pages 290–295. CSREA Press, 2005.

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

[5] N. Gupta, A. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of 15th IEEE International Conference on Automated Software Engineering, Grenoble, France*, pages 219–228, September 2000.

[6] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

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

[9] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

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

[11] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Chicago, IL, USA, July 12-16, 2003*, volume 2724 of *Lecture Notes in Computer Science*, pages 2488–2500. Springer-Verlag, 2003.

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

[13] F. Martin McNeill and Ellen Thro. *Fuzzy Logic: A Practical Approach*. Morgan Kaufmann, 1994.

[14] Christoph C. Michael, Gary McGraw, and Michael A. Chatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[15] M. Holcombe P. McMinn. Evolutionary testing of state-based programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Seattle, WA, USA, June 26-30, 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1363–1374. Springer-Verlag, 2005.

[16] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(3):263–282, September 1999.

[17] Paolo 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.

[18] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. *ACM SIGSOFT Software Engineering Notes*, 23(2):73–81, March 1998. ISSTA 98: Proceedings of the ACM SIGSOFT Internatioal Symposium on Software Testing and Analysis.

[19] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, Washington DC, USA, 25-29 June 2005. ACM Press.

[20] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.