Improving Evolutionary Class Testing in the Presence of Non-Public Methods

Stefan Wappler
Technical University of Berlin
DaimlerChrysler AITI
Ernst-Reuter-Platz 7, D-10587 Berlin, Germany
Phone: +49 30 314 78419

stefan.wappler@tu-berlin.de

Ina Schieferdecker
Technical University of Berlin
Sekretariat FR5-14
Franklinstr. 28/29, D-10587 Berlin, Germany
Phone: +49 30 3463 7241

ina@cs.tu-berlin.de

ABSTRACT

Automating the generation of object-oriented unit tests is a challenging task. This is mainly due to the complexity and peculiarities that the principles of object-orientation imply. One of these principles is the encapsulation of class members which prevents non-public methods and attributes of the class under test from being freely accessed.

This paper suggests an improvement of our automated search-based test generation approach which particularly addresses the test of non-public methods. We extend our objective functions by an additional component that accounts for encapsulation. Additionally, we propose a modification of the search space which increases the efficiency of the approach. The value of the improvement in terms of achieved code coverage is demonstrated by a case study with 7 real-world test objects. In contrast to other approaches which break encapsulation in order to test non-public methods, the tests generated by our approach inherently guarantee that class invariants are not violated. At the same time, refactorings of the encapsulated class members will not break the generated tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Test coverage of code, Testing tools*

General Terms

Verification

Keywords

unit testing, object-orientation, encapsulation, evolutionary testing

1. INTRODUCTION

The automation of test generation has been attracting many researchers due to the benefits it promises in terms of cost saving and test quality improvement. Various approaches to structure-oriented test generation for object-oriented class testing have been suggested [2, 11, 12, 13, 18, 6, 10, 15], tool support is also available on the market (e.g. [1, 3, 8]). The approaches aim at generating a preferably compact set of tests that achieve high structural coverage of the source code of the class under test.

However, existing approaches are limited in the presence of non-public methods: either, they break the encapsulation of the class under test and directly access non-public data and methods to increase code coverage; or they do not address the test of non-public methods at all, meaning that they test non-public methods only partially (as a by-product of the tests of the public methods). Breaking encapsulation during testing appears tempting since it greatly facilitates the setting of object states, the observability of object behavior, and the invocation of non-public methods – and hence the achievement of high code coverage. On the other hand, it can lead to inconsistent objects which do not comply with their class invariants. Then objects are tested which can never occur when the class under test is integrated into an application and is used via its public interface only. The degree of achieved code coverage is deceptive in this case. In addition, tests involving breaking of encapsulation are susceptible to be broken when class refactorings, such as the renaming of a private attribute, are performed. To the best of our knowledge, related work in the area of object-oriented unit test generation does not explicitly discuss covering non-public methods without breaking encapsulation.

This paper proposes an improvement of our approach to automatic test generation for class testing. It extends our previous work in that objective functions are defined also for test goals (i.e. uncovered code elements) belonging to non-public methods, as opposed to the original approach where objective functions were defined for test goals of public methods only. This enables the increase of code coverage for non-public methods by solely using the public class interface. Additionally, a new representation of test sequences is suggested that removes the separation of the search space into method call sequence space and parameter space. This enables the application of a much simpler search algorithm.

This paper is organized as follows: Section 2 recapitulates our evolutionary class testing approach, Section 3 describes the improvements of the approach, Section 4 reports on the

experiments performed to examine the improvements, and finally Section 5 concludes the paper.

2. EVOLUTIONARY CLASS TESTING

Evolutionary structural testing [4, 9, 16] is a search-based approach to software test generation. Its main idea is to transform the task of creating tests that achieve high structural coverage of the source code of the unit under test to a set of optimization problems, each of which an evolutionary algorithm tries to solve. Each uncovered code element (e.g. in the case of branch testing each branch of the control flow graph of the software under test) becomes a *test goal* for which a covering test is sought by heuristically exploring the space of all conceivable tests. An evolutionary algorithm is a meta-heuristic optimization technique that relies on the principles of selection, variation, and the survival of the fittest, according to the Darwinian theory of evolution. It maintains a set of candidate solutions to the search problem at hand and iteratively improves the single solutions by selecting promising solutions based on their *fitness*, combining them and mutating them to obtain new solutions that are potentially better with respect to the optimization problem. This procedure of selection and variation is repeated until a predefined termination criterion applies, e.g. the maximum allowed number of iterations of the evolutionary cycle is exceeded or a sufficiently good solution has been encountered. The fitness of a candidate solution originates from the *objective function* which numerically expresses the suitability of a candidate.

Evolutionary class testing (ECT) [15, 14] applies genetic programming [5], a particular evolutionary algorithm, to generate *test sequences*, i.e. sequences of method calls that create objects, manipulate their states, and call the method to be examined. ECT uses a tree-based representation of test sequences in order to account for the method call dependences and thus to minimize the occurrence of unexecutable test sequences. For each test goal, ECT defines an individual objective function. This objective function measures the distance between the targeted test goal and the execution flow produced when a candidate test sequence is being executed (an instrumented version of the source under test is used to allow for execution flow comprehension). ECT composes the following three distance metrics for the definition of an objective function: method call distance, approach level, and branch distance. Method call distance accounts for the premature termination of the execution of a candidate test sequence due to a runtime exception. It relates to the number of unexecuted methods. Approach level refers to the number of potential problem nodes that lay on the shortest path from the actual problem node to the targeted test goal. A problem node with respect to a test goal is a node of the control flow graph at which execution can diverge down a branch that prevents the test goal from being attainable. Branch distance relates to the predicate assigned to the actual problem node; it expresses how "far" the evaluation of the predicate was away from being evaluated to the opposite boolean outcome.

3. EXTENSIONS

We suggest two improvements to the evolutionary class testing approach: a major improvement concerning objective function design in order to address testing non-public methods, and a minor improvement concerning the underlying representation in order to simplify the evolutionary search algorithm.

3.1 Covering Non-Public Methods

Generating a test sequence that covers a test goal belonging to a non-public method means generating a test that indirectly invokes that non-public method with both suitable arguments (if any required) and suitable object states. To this end, the search for a covering test sequence can be decomposed into two subtasks: first, test sequences are sought which invoke the non-public method. Such an invocation might be control-dependent on several conditions; therefore, the search must first succeed in satisfying the control-dependent conditions favorably. Second, once test sequences invoking the non-public method under question are encountered, the search must concentrate on satisfying the conditions of the non-public method on which the test goal is control-dependent favorably. We perform a static analysis to identify all statements of the public methods of the class under test which involve a call to the non-public method under question. These statements will be referred to as *call points*. An individual evolutionary search is carried out for each call point. We construct the objective function for a test goal of a non-public method in such a way that it rewards test sequences that cover the call point. If the call point is not covered by a candidate test sequence, the three distance metrics are calculated with respect to the call point. In addition, a constant penalty τ is added to the objective value that indicates that the call point has not been reached. Otherwise, if the call point has been reached but the test goal was not attained, the distance metrics approach level and branch distance are calculated with respect to the test goal.

$$\omega(c, g) = \begin{cases} \lambda d_{AL}(g) + d_B(g) & c \ reached \\ \lambda d_{MC}(c) + d_{AL}(c) + d_B(c) + \tau & otherwise \end{cases}$$

(1)

Equation 1 shows the objective function ω for call point c and test goal g, where d_{MC} is the method call distance, d_{AL} is the approach level, d_B is the branch distance, and τ is a constant penalty. The value of τ must be chosen in such a manner that it is greater than the greatest possible value of $d_{AL}(g) + d_B(g)$. The resulting objective function also accounts for runtime exceptions in correspondence with [14].

In the case that a private method is called only by another private method, chains of call points result from the static analysis. The objective function must account for such a chain by rewarding test sequences that cover it. We suggest adding the penalty τ multiple times, depending on the length of the uncovered chain.

3.2 Improved Representation Design

In our previous work, we employed a hybrid evolutionary algorithm: on the first search level, test sequence fragments (method call sequences without arguments) were sought, while on the second level, for each candidate test sequence fragment the corresponding parameter values were sought. However, this hybrid search algorithm is expensive: a completely new parameter search is carried out for each fragment, good parameter values from previous parameter searches are not

reused. In order to simplify the search and thus make it more economic, we suggest a modification of the representation of test sequences: primitive values are now also part of a method call tree, meaning that the method call dependence graph also includes primitive types; methods can be call-dependent on a primitive type then. Additionally, we introduce the type *selector*; each class-type argument of a method implies an additional call dependence on the selector type. Values of this type are positive integers which are interpreted as parameter object selectors (refer to [15] for this concept). The value range of a selector value is adapted to the actual range of candidate objects using the *modulo* operator. Figure 1 exemplifies the new representation. At
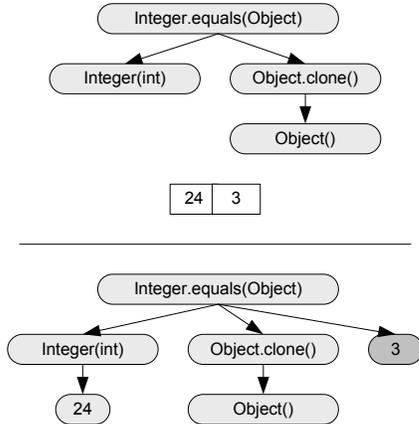

Figure 1: Method call trees

the top, it shows a method call tree representing a test sequence fragment (point in sequence space). Together with the vector $(24, 3)$ (point in parameter space), the fragment will be completed to the following test sequence:

```
Integer i1 = new Integer(24);
Object o1 = new Object();
Object o2 = o1.clone();
boolean b = i1.equals(/*3-->*/ o2);
```

Note that the parameter selector value 3 (second vector element) is mapped to instance o2 since o2 is the 3rd candidate parameter object for the call to method `equals`. At the bottom, the figure shows a method call tree that already includes the parameter information and represents the same test sequence as the fragment and the vector together (point in the unified search space).

4. EXPERIMENTS

We implemented the evolutionary class testing approach in the tool EvoUnit and used it to carry out a case study with 34 real-world classes taken from five different open-source Java projects. Due to space restrictions, we present the results of only 7 test objects which are supposed to be a representative subset. Table 1 lists the test objects along with the results achieved by EvoUnit. EvoUnit uses the genetic programming system ECJ [17], which was configured as follows: 50 candidate solutions in 1 global population, uniform tree initialization, tournament selection (tournament size 7), subtree crossover, ERC mutation, demotion mutation, promotion mutation, pure reinsertion, 200 generations at most.

The first group of classes is taken from the Java 2 SDK 1.4.2, the second group from the Quilt project 0.6a5, and class `Range` from project JFreeChart 1.0.1. Column *ELOC*

Test object	ELOC	b_T	b_N	Cov	Evals
CodeSource	236	110	70	87.0%	152853
LinkedList	317	68	11	98.3%	27398
StringTokenizer	108	29	17	93.7%	23299
TreeSet	108	27	3	92.7%	19803
Directed	137	25	5	82.0%	45558
Exit	30	7	1	100.0%	10
Vertex	90	23	1	87.0%	30000
Range	144	42	4	97.6%	10006

Table 1: Test objects and EvoUnit results

gives the number of executable lines of code. Column b_T shows the total number of branches of the class, while column b_N shows the number of branches of the non-public methods. Column *Cov* shows the branch coverage achieved by EvoUnit, whereas column *Evals* shows the number of objective function evaluations needed. The last two columns are averages over 50 independent runs. Figure 2 contrasts EvoUnit (light-gray bars) with a random test sequence generator (dark-gray bars). It shows the achieved coverages by both approaches and indicates the statistical significance of the differences using stars: no star means the difference is
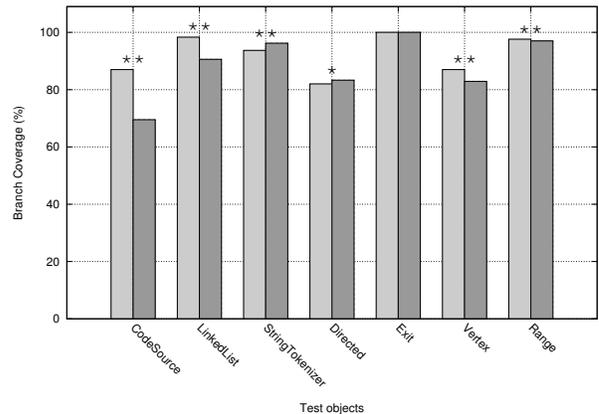

Figure 2: EvoUnit vs. Random Testing

due to change, one star means the difference is real with a probability of 0.95%, and two stars mean the difference is real with a probability of 0.99%. The results suggest that EvoUnit achieved higher coverages than the random approach in average.

Figure 3 suggests the effectiveness of the improvement concerning the coverage of non-public methods. It shows the results from table 1 (light-gray bars), contrasted with the results obtained with EvoUnit without the improvement (dark-gray bars). The shown coverages refer to the non-public methods of the test objects and are also avered over 50 runs.
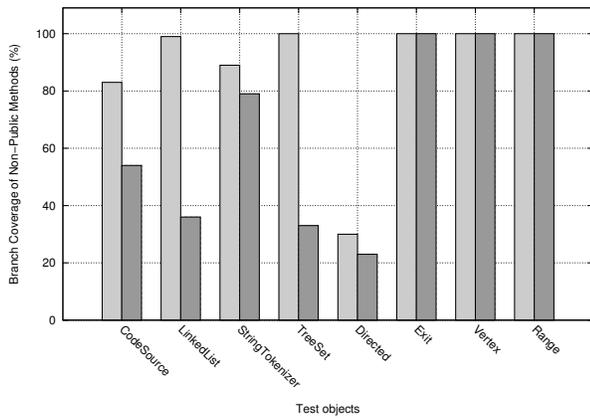
Figure 3: Improved EvoUnit vs. original EvoUnit

5. CONCLUSION AND FUTURE WORK

This paper discussed an improvement of our evolutionary class testing approach which addresses the test of non-public methods. An extension to the design of the objective functions was suggested that guide the evolutionary searches to find test sequences that indirectly cover non-public methods. The generated tests use the public class interface only and do not break encapsulation, hence ensuring both the legality of the participating objects and the maintainability of the tests. Additionally, a new representation of test sequences was discussed which unifies the sequence search space and the parameter search space in order to simplify the searches and hence increase the efficiency.

Future work includes the experimentation with more test objects, a more detailed analysis of the results (e.g. why is random testing sometimes better than evolutionary testing), further improvements of the distance functions, and the adaptation of the chaining approach [7] to address the state problem more effectively.

Acknowledgement

6. REFERENCES

[1] Agitar Software, Inc. Agitator. http://www.agitar.com, 2006.

[2] S. Beydeda and V. Gruhn. Bintest – binary search-based test case generation. In *Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society Press, 2003.

[3] Instantiations, Inc. CodePro. http://www.instantiations.com, March 2007.

[4] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic test data generation using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[5] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[6] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of object-oriented programs. In *MIC2005: The Sixth Metaheuristics International Conference*, September 2005.

[7] P. McMinn and M. Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64, 2006.

[8] Parasoft, Inc. Jtest. http://www.parasoft.com.

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

[10] K. Sen and G. Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, Lecture Notes in Computer Science. Springer, 2006. (To Appear. Tool Paper).

[11] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[12] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.

[13] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, New York, NY, USA, 2005. ACM Press.

[14] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI-2006)*, pages 3193–3200, Vancouver, Canada, July 2006. IEEE Press.

[15] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 2006 conference on Genetic and evolutionary computation*, pages 1925–1932, New York, NY, USA, 2006. ACM Press.

[16] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(1):841–854, 2001.

[17] G. C. Wilson, A. McIntyre, and M. I. Heywood. Resource review: Three open source systems for evolving programs: Lilgp, ecj and grammatical evolution. *Genetic Programming and Evolvable Machines*, 5(1):103–105, 2004.

[18] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.