On Test Data Generation of Object-Oriented Software

Andrea Arcuri and Xin Yao (Advisor)
The Centre of Excellence for Research in
Computational Intelligence and Applications (CERCIA)
The School of Computer Science, The University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
{A.Arcuri,X.Yao}@cs.bham.ac.uk

Abstract

Nowadays, Object-Oriented (OO) languages are widely used in the development of many different kinds of applications. However, testing those applications is still very expensive and time-consuming for the software community. The automation of this task would therefore be highly desirable. Although automatic testing of procedural software has been studied in depth for many years, comparatively little work has been done about OO software. Different techniques exist. However, the most promising one is probably to model the task as a Search Problem.

This paper explains why automatic testing of OO software is more difficult than procedural software. These difficulties provide strong challenges to the Natural Computation and Software Engineering communities. A brief review of the literature of the subject follows. The issues of the current State-of-Art of the field are then outlined. Finally, some open research problems are discussed.

1 Introduction

Software Testing is used to find the presence of bugs in computer programs [23]. If no bugs are found, testing cannot guarantee that the software is bug-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software developing [4]. This cost is paid because Software Testing is very important. Releasing bug-ridden, non-functional software is an easy way to lose customers. Besides, bugs should be discovered as soon as possible, because fixing them when the software is in a late stage of development is much more expensive than at the beginning. For example, in the U.S.A it is estimated that every year around $20 billion could be saved if better testing was done before releasing new software [27]. Therefore, a system that can automatically generate tests could save billions for the software industry.

Among the adequacy criteria, *White Box Testing* is one of the most widely known. It depends only on the source code. A formal specification of the software is not needed. Given a coverage criterion (e.g., statements coverage), the problem of *Test Data Generation* consists in finding a suitable set of data that results in the highest possible coverage when given as input to the functions under test. This data is used to create *drivers* that call the functions under test with them as input, and the results are then compared against an *Oracle*.

Different techniques have been developed to automate this task (e.g., *Symbolic Execution* [16]). For a broad survey see [19, 13]. However, the one that seems to be the most promising is to model the task as a *Search Problem* [14, 10]. In such a case, well known metaheuristics like Genetic Algorithms (GAs) [15] can be employed to solve the problem. GAs have been widely used with promising results. However, in the Search Based Software Engineering community there seems to be a strong bias in favour of the GAs. This bias may not be warranted, considering that no search algorithm is better than any other over all search problems [38]. Hence, because Search Based Software Engineering is a new area of research, several search algorithms should be analysed and compared, rather than just using the most popular one. In the recent years, in fact, other metaheuristics like Memetic Algorithms (MAs) [21] and Estimation of Distribution Algorithms (EDAs) [22] have been reported to achieve better results in some empirical experiments [33, 2, 24]. However, the Search Based Software Engineering field is still too young (i.e., not enough research has been done yet) to decide which is the best search algorithm that should be applied to it.

Although much research has been done on the automation (not only the search based one) of Test Data Generation of procedural software, little is known about the testing of Object-Oriented (OO) software with search based tech-

niques. This is a problem, because "academic research in software testing can and does have relevance to industrial applications" (M. Boshernitsan et al. [5], 2006). At the moment, a system that can generate an optimal set of tests for any generic OO program in an automatic way has not been developed yet. Many issues still need to be addressed and solved. Therefore, more research in this field is needed.

Section 2 describes the difficulties of automating the testing of Object-Oriented Software. A brief survey on the current state of the art of this field follows in Section 3. Section 4 shows the open problems that still need to be solved and future research directions. Finally, the conclusion of the paper can be found in Section 5.

2 Testing OO Software

Testing Object-Oriented Software (OOS) requires more efforts than testing Procedural Software (PS) [3]. This happens because special features of the OO paradigm like *encapsulation*, *inheritance* and *polymorphism*, make the testing phase harder. A typical example is when a new subclass is introduced: all the inherited methods of the superclass need to be retested again in the new context. In this paper we focus on Test Data Generation (TDG), i.e., our aim is to find a test suite that gives the best coverage. Without loss of generality, we consider Java as an example of OO language and C as an example of procedural language.

The *State Problem* in PS testing [20] is hard to handle. Functions can have an internal state (e.g., static variables in the C programming language) that might need to be put in a particular configuration if we want to cover a precise branch. If the system does not have a direct access to this state, a sequence of function calls is needed to put the internal state in the wanted configuration. How to find this sequence is a difficult task that complicates the testing. In OOS there is the same problem, but usually the states are more complex (e.g., Red-Black Trees). Hence, simple techniques that might reasonably work for PS will not be enough for handling OOS. More sophisticated techniques need to be developed. However, it is important to outline that it is possible to have very complicated states in PS. Furthermore, we can use OO languages like Java for creating PS style programs (e.g., all static functions with no internal state and no objects at all). At any rate, usually this does not happen. A statistical analysis of a very large amount of software would help to support this claim, but it is beyond the scope of this paper.

Information Hiding, besides complicating the state problem, also makes the choice of the input parameters for the tested function more difficult. In fact, these parameters can be object references that might need to be put in a particular state. If the state is hidden, we need to call some functions on these parameters to put them in the desired state. This is an issue that does not arise in PS. In PS, a function's input might be a very complex data structure, but its state is open and easier to manipulate [31]. Besides, it might happen that an object cannot be directly instantiated (e.g., if the *singleton* design patter is used) or it is necessary that such an object instance has to be returned by a function call (e.g., that function belongs to another class which uses its hidden internal state and/or internal private classes to initialise the new object instance). Again, this does not happen in PS. Also covering branches in private methods is not easy. In fact, several public methods may lead to the execution of the same private function, and a search through these methods and their input parameters needs to be done, because maybe only one of these methods calls the private one with the right parameters.

Polymorphism is widely used in OO software. Unfortunately, this makes the exploitation of static source code analyses difficult (if not impossible). Besides, it enlarges the search space of the input parameters of the functions. E.g., if a Java function takes in input a reference to the abstract class "Object", all the classes in the system and in all the support libraries are possible candidates.

The presence of external libraries, the source code of which is not available, is a problem for testing in any type of software. Statistically showing that this situation arises more often in OOS is beyond the scope of this paper. However, it is not rare that the class under test *extends* an external class. This may make the automation of the testing phase very hard.

Other features of the OO paradigm (e.g., templates and exceptions) may complicate the testing even further.

3 The State of Art

Different experimental tools have been developed to automatically test OO software. The early ASTOOT [12] generates tests from algebraic specifications of the functions. Korat [6] and TestEra [18] use isomorphic generation of data structures, but need predicates that represent constraints on these data structures. Rostra [39] uses bounded exhaustive exploration with concrete values. On the other hand, tools that exploit the *symbolic execution* [16] include for example Symstra [40], Symclat [11], the work of Buy et al. [7] and the model-checker Java PathFinder used for test data generation [29]. Although promising results have been reported, these techniques are unlikely to scale well when handling complex data structures is required [30]. Besides, as clearly stated in [11], at the moment they have difficulties in handling non-linear predicates, non-primitive data types, loops, arrays, etc.

Due to all these limitations, the use of search based optimisation techniques for testing OO programs has started to be investigated in the last few years. Tonella [28] used Ge-

netic Algorithms for generating unit tests of Java programs. Solutions are modelled as sequences of function calls with their inputs and caller (an object instance or a class if the method is static). Special crossover and mutation operators are proposed to enforce the feasibility of generated solutions. Empirical tests were carried out on six classes taken from the Java API.

Similar work with GAs has been done by Wappler and Lammermann [34], but they used standard evolutionary operators. This can cause the generation of infeasible individuals, which will be penalised by the fitness function. Besides, they investigated the idea of separately optimising the parameters, the function calls and the target instanced objects. Coverage tests were done on a class comprising 27 lines of code. Three classes from the Java API were used to test their criteria for evaluating the inconvertible individuals. However, they did not provide any coverage measurement.

Strongly Typed Genetic Programming (STGP) has been used by Wappler and Wegener [36] for testing Java programs. Tests were carried out on four different container classes. They extended their approach by considering the problem of the raised exceptions during the evaluation of a sequence [35]. If an exception is thrown, the fitness will consider how distant the method in which it is thrown is from the target method in the test sequence. Their test cluster had less than 50 lines of code.

In his master's thesis [26], Seesing investigated the use of STGP as well. Experiments were carried out on five different classes.

Liu et al. [17] used a hybrid approach, in which Ant Colony Optimisation is exploited to optimise the sequence of function calls. A Multi-agent Genetic Algorithm is used then to optimise the input parameters of those function calls. Empirical tests were carried out on two data structure classes.

Also Cheon et al. [9] proposed an evolutionary tool, but they implemented and tested only a random search. No empirical tests were reported. They proposed to exploit the specification of the functions that return boolean values to improve the fitness function [8]. Again, no details of the empirical tests were reported.

Our work [1, 2, 25] has focused on the testing of *Container Classes* (e.g., Vector, Stack and Red-Black Tree). Besides analysing how to apply different search algorithms (Random Search, Hill Climbing, Simulated Annealing, Genetic Algorithms, Memetic Algorithms and Estimation of Distribution Algorithms) to the problem and exploiting the characteristics of this type of software to help the search, we introduced more general techniques that can be applied to any OO software. E.g., we proposed an improved branch distance that solves an issue regarding the evaluation of conjunctions of predicates [2]. Our test cluster was composed of up to seven container classes, for a total of over 5000 lines of code.

There are several issues in the current State of Art of Test Data Generation for Object-Oriented Software:

- Only little work has been done using optimisation algorithms.
- Empirical tests have always been done on very small clusters of classes. This reduces the reliability of the results.
- There is no common benchmark cluster on which different authors can test and compare their techniques. This makes it difficult to evaluate the performance of a novel technique against existing ones.
- Apart from Random Search, usually there are no comparisons between different optimisation algorithms on the testing of the same classes.
- There is no theoretical work on test data generation for OO software. All articles are of empirical nature.

A commercial tool that automatically generates unit tests for Java programs is *Agitator* [5]. At the moment of writing this paper, it does not use any optimisation algorithm. It would be interesting to know with which degree it is successfully used by its customers. However, some big companies might have developed their own in-house automatic testing tool (e.g., Daimler-Chrysler). As it is true that industrial products are influenced by academic research [5], it is true as well that the academics can take advantages from these industrial experiences. In fact, academics need to know what are the real world problems and their difficulties to understand what are the important issues that need to be solved.

4 Future work

Test Data Generation is a vast subject that has not been sufficiently explored. This is particularly true in the case of Object-Oriented (OO) software. Although many open research questions are still in need of an answer, we found three points of particular interest that we want to investigate at the moment:

- In procedural software, the use of local search algorithms (e.g., Hill Climbing) has often been ignored because the search landscape has always been considered too "complex, discontinuous, and non-linear" (J. Wegener et al. [37], 2001). Although the test clusters were relatively small, recent research shows that exploiting local search gives good results [33, 1, 2]. This leads us to investigate different types of local search

and how to integrate them in global search algorithms such as Memetic Algorithms.

- Although they were considered as one of the main points to be addressed [14], analyses of fitness landscapes have not received much attention in the Search-Based Software Engineering community (we are aware only of one example [32]). We think that such analyses are important to obtain a deeper understanding of the problem. This will help us to design and tune novel, more efficient search algorithms for the Test Data Generation task. A comparison between the search landscapes of procedural and OO software will help us understand if the techniques that have been studied and applied for many years on the former will be still valid for the latter. A priori, we know that OO software is more difficult to test, but we do not know how much more difficult it is. Numerical analyses of the search landscapes may help to answer this question. Besides, it is interesting to investigate whether different typologies of OO software (e.g., data structure, math and GUI applications) manifest significant diversities in their relative search landscapes.

- Scalability is a factor that has not been sufficiently considered. At this stage, we simply do not know whether any of the algorithms described in the literature will be able to scale up to industrial-size software. This might be a problem for unit testing, and furthermore for integration and system testing.

We claim that these research questions are of the utmost importance. Answering them would significantly help the Software Testing community to better understand the characteristics of the problem of Test Data Generation. This would be a first step that would take us closer to the goal of completely automating the testing of software.

5 Conclusion

In this paper, we have shown the difficulties of testing Object-Oriented software. Because it is more difficult than procedural software, plenty of open problems still need to be addressed and solved. Although this seems to offer ideal conditions for doing research, only little work that uses search based optimisation algorithms is known on Test Data Generation for Object-Oriented Software. We believe that this situation will drastically change in the next few years, because we think that other techniques such as symbolic execution are not very suitable for OO software. However, hybrid systems that combine optimisation algorithms with more traditional techniques may result in better performances.

In the research community there is a lack of organisation on how to evaluate and compare the results of novel proposals against existing ones. This is damaging, because when a new technique is proposed, it is often difficult to understand if it is really better than previous techniques. Hence, deciding to follow up and try to improve a novel technique can result in a waste of resources if it was not as good as it was described. For example, search based techniques seem better than symbolic execution, but no work that compare these techniques is known to us. Furthermore, Genetic Algorithms are rarely compared to other metaheuristics. Although it is reasonable that, at the moment, the Software Engineering problems are too difficult to be solved from a theoretical point of view, there is no excuse for the lack of any sort of empirical verification among the different systems.

Among the several open problems that need to be addressed, we described three of them that we found particularly interesting. We believe that they are challenging issues that, if solved, could help the Software Testing community and the Software Industry.

6 Acknowledgements

This work is supported by an EPSRC grant (EP/D052785/1). The authors wish to thank Thomas Miconi for having proofread an early draft of this paper.

References

[1] A. Arcuri and X. Yao. Search based testing of containers for object-oriented software. Technical Report CSR-07-3, University of Birmingham, 2007.

[2] A. Arcuri and X. Yao. A memetic algorithm for test data generation of object-oriented software. In *Congress on Evolutionary Computation (CEC)*, 2007 (to appear).

[3] S. Barbey and A. Strohmeier. The problematics of testing object-oriented software. In *Second Conference on Software Quality Management*, pages 411–426, 1994.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, 1990.

[5] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–180, 2006.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[7] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 39–48, 2000.

[8] Y. Cheon and M. Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1952–1954, 2006.

[9] Y. Cheon, M. Y. Kim, and A. Perumandla. A complete automation of unit testing for java programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295, 2005.

[10] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[11] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

[12] R. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, pages 101–130, 1994.

[13] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linkoping*, pages 21–28, 1999.

[14] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.

[15] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.

[16] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

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

[18] D. Marinov and S. Khurshid. Testera: A novel framework for testing java programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2001.

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

[20] P. McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield, 2005.

[21] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report 826*, 1989.

[22] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions i. binary parameters. In *Proceedings of The Fourth International Conference on Parallel Problem Solving from Nature*, pages 178–187, 1996.

[23] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[24] R. Sagarna. *An Optimization Approach for Software Test Data Generation: Applications of Estimation of Distribution Algorithms and Scatter Search*. PhD thesis, University of the Basque Country, 2007.

[25] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *Congress on Evolutionary Computation (CEC)*, 2007 (to appear).

[26] A. Seesing. Evotest: Test case generation using genetic programming and software analysis. Master's thesis, Delft University of Technology, 2006.

[27] G. Tassey. The economic impacts of inadequate infrastructure for software testing, final report. *National Institute of Standards and Technology*, 2002.

[28] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[29] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[30] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[31] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 149–160, 2002.

[32] H. Waeselynck, P. T. Fosse, and O. A. Kaddour. Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering*, 12(1):35–63, 2006.

[33] H.-C. Wang and B. Jeng. Structural testing using memetic algorithm. In *Proceedings of the Second Taiwan Conference on Software Engineering*, 2006.

[34] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1053–1060, 2005.

[35] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Congress on Evolutionary Computation (CEC)*, pages 851–858, 2006.

[36] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1925–1932, 2006.

[37] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[38] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[39] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.

[40] 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*, pages 365–381, 2005.