Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming

José Carlos Bregieiro Ribeiro
Polytechnic Institute of Leiria
Morro do Lena, Alto do Vieiro
Leiria, Portugal
jose.ribeiro@estg.ipleiria.pt

ABSTRACT

In evolutionary testing, meta-heuristic search techniques are used to generate high-quality test data. The focus of our on-going work is on employing evolutionary algorithms for the structural unit-testing of object-oriented Java programs.

Test cases are evolved using the Strongly-Typed Genetic Programming technique. Test data quality evaluation includes instrumenting the test object, executing it with the generated test cases, and tracing the structures traversed in order to derive coverage metrics. The strategy for efficiently guiding the search process towards achieving full structural coverage involves favouring test cases that exercise problematic structures and control-flow paths. Static analysis and instrumentation is performed solely with basis on the information extracted from the test objects' Java Bytecode.

Relevant contributions include the introduction of novel methodologies for automation, search guidance and input domain reduction, and the presentation of the *eCrash* automated test case generation tool.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Verification

Keywords

Search-Based Test Case Generation, Evolutionary Testing, Strongly-Typed Genetic Programming, Object-Orientation

1. INTRODUCTION

Software testing is expensive, typically consuming roughly half of the total costs involved in software development while adding nothing to the raw functionality of the final product [1]. Yet, it remains the primary method through which confidence in software is achieved. A large amount of the resources spent on testing are applied on the difficult and time consuming task of locating quality test data; automating this process is vital to advance the state-of-the-art in software testing.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
ACM 978-1-60558-131-6/08/07.

Test data generation deals with locating good test data for a particular test criterion; the application of evolutionary algorithms to this process is often referred to as *evolutionary testing* or *search-based test case generation*. In evolutionary testing, meta-heuristic search techniques are employed to select or generate test data. The search space is the input domain of the test object, and the problem is to find a set of test cases that satisfies a certain test criterion.

The focus of our on-going work is that of employing evolutionary algorithms for generating and evolving test cases for the structural unit-testing of third-party object-oriented (OO) Java programs. In the following Section, we start by introducing the concepts underlying our research; in the Technical Approach section, we outline our methodology. In Section 4 the experiments performed and results obtained so far are described. Topics for future work are proposed in Section 5, and the concluding Section presents some final considerations and the most relevant contributions.

2. BACKGROUND AND TERMINOLOGY

When performing *unit-testing*, the goal is to warrant the robustness of the smallest units – the *test objects* – by testing them in an isolated environment. Unit-testing is performed by executing the test objects in different scenarios using relevant and interesting *test cases*.

A *test set* is said to be adequate with respect to a given criterion if the entirety of test cases in this set satisfies this criterion. Traditional structural adequacy criteria include branch, data-flow and statement coverage; the basic idea is to ensure that all the control elements in a program are executed by a given test set, providing evidence of its quality. The metrics for measuring the thoroughness of a test set can be extracted from the structure of the target object's source code, or even from compiled code (e.g. Java Bytecode).

The observations needed to assemble the metrics required for the evaluation of test data suitability can be collected by abstracting and modelling the behaviours programs exhibit during execution; for this purpose, the definition of an underlying model for program representation – usually a *control-flow graph* (CFG) – is generally required. *Dynamic analysis* involves executing the actual test object and monitoring its behaviour. Dynamic monitoring of structural entities can be achieved by *instrumenting* the test object, and *tracing* the execution of the structural entities traversed during test case execution. Instrumentation is performed by inserting probes in the test object.

For object-oriented programs, classes and objects are typically considered to be the smallest units that can be tested

in isolation. An object stores its state in fields and exposes its behaviour through methods. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* – a fundamental principle of object-oriented programming. One of the most pressing challenges faced by researchers in the evolutionary testing area is precisely the *state problem*, which occurs with objects that exhibit state-like qualities by storing information in fields that are protected from external manipulation – and that can only be accessed through the public methods that expose the classes' internals and grant the access to the objects' state.

A unit-test case for OO software consists of a *method call sequence*, which defines the test scenario. During test case execution, all participating objects are created and put into particular states through a series of method calls. Each test case focuses on the execution of one particular public method – the *method under test* (MUT). It is not possible to test the operations of a class in isolation; testing a single class involves other classes, i.e. classes that appear as parameter types in the method signatures of the *class under test* (CUT). The transitive set of classes which are relevant for testing a particular class is called the *test cluster*.

For search-based test case generation problems, the search space is the *input domain* of the test object. In the particular case of OO programs, the input domain encompasses the *parameters* of the test object's public methods – including the implicit parameter (i.e. the `this` parameter) and all the explicit parameters. As such, the goal of the evolutionary search is to find method call sequences that define interesting state scenarios for the variables which will be passed, as arguments, in the call to the method under test, so as to enable the full structural coverage of the test object.

Method call sequences can effectively be encoded using *Strongly-Typed Genetic Programming* (STGP). STGP was proposed by Montana [5] with the intention of addressing the "closure" limitation of the Genetic Programming technique. Closure means that any element can be a child node in a parse tree for any other element without having conflicting data types, which can lead to the generation of syntactically illegal trees. In contrast, STGP allows the definition of types for the variables, constants, arguments and returned values; the only restriction is that the data type for each element must be specified beforehand. This causes the initialization process and the various genetic operations to only construct syntactically correct trees.

The STGP search space is the set of all legal parse trees – i.e. all of the functions have the correct number of parameters of the correct type. STGP is thus particularly suited for representing method call sequences, as it enables the reduction of the search space to the set of *compilable* sequences, by allowing the definition of constraints that eliminate invalid combinations of operations. The usage of STGP in this context was first proposed in [14].

Still, syntactically correct and compilable method call sequences can result in *unfeasible* test cases. Such test cases abort prematurely during execution because a runtime exception is thrown; when this happens, it is not possible to trace the structural entities traversed because the final instruction of the method call sequence – i.e. the call to the method under test – is not reached. In contrast, *feasible* test cases are successfully executed and terminate with a call to the method under test.

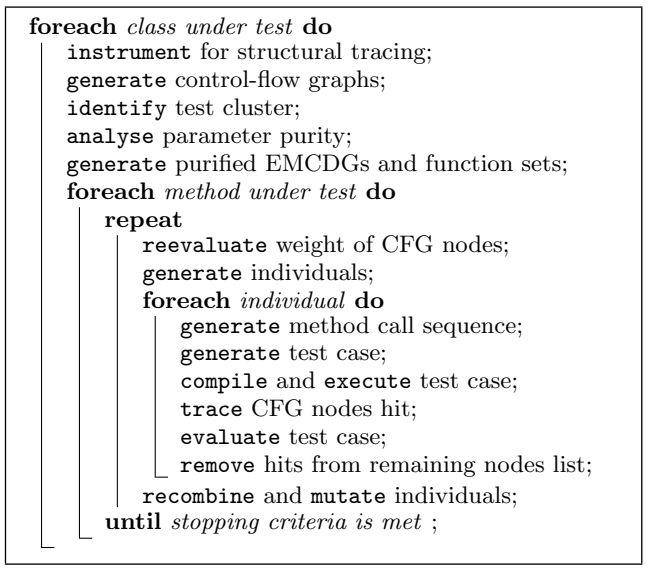

Figure 1: Methodology overview.

The massive number of distinct method call sequences that can possibly be created while searching for a particular test scenario constitutes a notorious hindrance to the test case generation process. *Input domain reduction* deals with the removal of irrelevant variables from a given test data generation problem, thereby reducing the size of the search space. The input domain can effectively be reduced by acknowledging that some methods in OO languages have no externally visible side effects when executed or, at least the extent of these side effects is limited in some way; these are called *pure methods* [11]. More interestingly, purity analysis is able to identify important purity properties even when a method is not pure, such as safe and read-only parameters: a parameter is *read-only* if the method does not write the parameter or any objects reachable from the parameter; a parameter is *safe* if it is read-only, and the method does not create any new externally visible paths in the heap to objects reachable from the parameter.

Purity analysis is especially useful in the context of search-based test case generation, as it provides a means to automatically identify and remove entries that are irrelevant to the search problem, reducing the size of the set of method calls from which the algorithm can choose when constructing the method call sequences that compose test cases.

3. TECHNICAL APPROACH

In this Section, our evolutionary approach for automatic test case generation is described. The concepts presented were implemented into the *eCrash* tool [7]. The algorithm depicted in Figure 1 overviews and summarizes the process.

3.1 Test Object Analysis and Instrumentation

The test object instrumentation and CFG generation tasks are performed statically with the aid of the Sofya framework [3], and must precede the test case generation and evaluation phases so as to allow tracing the structural entities traversed during test case execution.

Probe insertion is executed on the Java Bytecode level,

and CFGs are built with basis on the high-level information extracted from the Bytecode of the test object. Given that the test object's source code is often unavailable, working with Java Bytecode allows broadening the scope of applicability of the *eCrash* automated testing tool; it can be used, for instance, to perform structural unit-testing on third-party Java components [6].

3.2 Function Set Definition

Potential solutions (i.e. test cases) are encoded as STGP individuals and method call sequences as STGP trees. Each tree subscribes to a function set, which defines the STGP nodes legally permitted in the tree. For modelling call dependences and defining the function set, a similar methodology to that proposed by Wappler and Wegener [13, 14] is used; namely, an Extended Method Call Dependence Graph (EMCDG) is employed.

Also, our input domain reduction strategy involves the removal of irrelevant edges from the EMCDG; this is performed by analysing the purity of parameters with the aid of the Soot Framework [12], and using this information to build the *purified EMCDG*. The function set is computed with basis on the purified EMCDG, so as to include only those entries that are relevant to the search [10].

3.3 Test Case Generation

For representing and evolving test cases, the Evolutionary Computation in Java (ECJ) package [4] is used. The first step involved in the generation of the test cases' source-code is the linearization of the STGP trees using a depth-first transversal algorithm. The tree linearization process yields the method call sequence; source-code generation is performed by translating method call sequences to test cases, using the method signature information encoded into each STGP node.

Whenever a test case "hits" a CFG node, that node is removed from the *remaining nodes list*; the search stops when there are no CFG nodes left to be covered or after a predefined number of generations [8].

3.4 Test Case Evaluation

The quality of a given test case is related to the CFG nodes of the MUT which are the targets of the evolutionary search at the current stage of the search process. Test cases that exercise less explored (or unexplored) CFG nodes and paths must be favoured, with the objective of attaining the primary goal of the test case generation process – finding a set of test cases that achieves full structural coverage of the test object. The issue of steering the search towards the traversal of interesting CFG nodes and paths was addressed by assigning weights to the CFG nodes; the higher the weight of a given node the higher the cost of exercising it, and hence the higher the cost of traversing the corresponding control-flow path. Additionally, the weights of CFG nodes are re-evaluated at the beginning of every generation; nodes which have been recurrently traversed in previous generations and/or lead to uninteresting paths are penalised.

At the beginning of each generation the weight of each CFG node is multiplied by: a *weight decrease constant* value α, so as to decrease the weight of all CFG nodes indiscriminately; a *hit count factor*, which worsens the weight of recurrently hit CFG nodes; a *path factor*, which improves the weight of nodes that lead to interesting nodes and belong to interesting paths. After being re-evaluated, the nodes' weights are normalized. For *feasible test cases*, the fitness is evaluated as being the average weight of the nodes covered by the test case. This computation is performed with basis on the trace information; relevant data includes the "Hit List" – i.e. the set of traversed CFG nodes. For *unfeasible test cases*, the fitness of the individual is calculated in terms of the distance between the index of the method call that threw the exception and the method call sequence's length; the higher the percentage of instructions executed, the higher the individual's quality. Also, an *unfeasible penalty constant* value β is added to the final fitness value, so as to penalise unfeasibility [9].

4. EXPERIMENTS AND RESULTS

This Section describes the empirical studies implemented with the objectives of validating the approach.

4.1 Test Case Evaluation

This experiment, described further detail in [9], was performed in order to verify the adequateness of our test case evaluation strategy, and to experiment with different configurations for the *probabilities of evolutionary operators* – mutation, reproduction and crossover – and the values of the *test case evaluation parameters* – the *weight decrease constant* α and the *unfeasible penalty constant* β.

We observed that our strategy allows unfeasible test cases to be considered at certain points of the evolutionary search – once the feasible test cases that are being bred cease to be interesting because they only traverse recurrently hit CFG nodes. A good compromise between the intensification and diversification of the search can be achieved, so as to favour the diversity and complexity of method call sequences – which are often needed to define elaborate state scenarios and enable the test set to achieve full structural coverage.

The parametrization that achieved the best results was that of assigning balanced probabilities to the evolutionary operators, low values to α (≈ 0.5), and a value of half the initial nodes' weight to the unfeasible penalty constant β.

4.2 Input Domain Reduction

A study on the impact of employing purity analysis was presented in [10], with encouraging results. For the classes employed in this experiment (`Stack`, `BitSet`, `BoolStack` and `ObjectVector` of JDK 1.4.2), the statistics show a clear reduction in the size of the input domain; the number of function set entries when purity analysis is used is, on average, 31.6% lower than that obtained when no purity analysis is employed.

The impact of this reduction on the test case generation process was also visible on the case studies performed on the `Stack` and `BitSet` classes using the *eCrash* tool. For the `Stack` class, the number of generations required to attain full coverage using purity analysis was, on average, only 34% of that required without purity analysis. For the `BitSet` class the improvement was not as clear; still, the average percentage of test cases that accomplished full coverage within a maximum of 100 generations increased approximatelly 6%.

5. FUTURE WORK

Several open problems persist in the area of search-based test case generation. Future work will be mainly focused on

addressing the challenges posed by the three cornerstones of OO programming: *encapsulation*, *inheritance*, and *polymorphism*. The importance of the inheritance and polymorphism properties, in particular, is yet to be fully studied by researchers in this area. Inheritance allows the treatment of an object as its own type or its base type; many types (derived from the same base type) can be treated as if they were one. Polymorphism means "different forms", and allows one type to express its distinction from another similar type, as long as they are both derived from the same base type; this distinction is expressed through differences in behaviour of the methods that can be called through the base class [2].

Search space sampling deals with the inclusion of all the relevant variables to a given test object into the test data generation problem, so as to enable the coverage of the entire search space whenever possible. Because the test cluster cannot possibly include all the subclasses that may override the behaviours of the classes which are relevant for the test object, adequate strategies for search space sampling – which take the commonality among classes and their relationships with each other into account – are of paramount importance.

In a more distant future, we also intend to experiment with parallel systems in order to enhance our methodology's performance. The evaluation procedure, in particular, is inherently parallelizable, as all the test cases generated in a given generation can be executed simultaneously. The development of a full and stable test tool has also been an underlying principle of our research since its beginning. Still, further work must still be put on the development of several features – such as the implementation of a Graphical User Interface – if the *eCrash* tool is to be employed in an industrial environment or by users unfamiliar with its intricacies.

6. CONCLUSIONS

This paper outlined our evolutionary approach to the automated generation of test cases for the structural unit-testing of object-oriented Java programs. Our our-going work already led to relevant contributions in the area, which include the introduction of novel methodologies for automation, search guidance and input domain reduction, and the presentation of the *eCrash* tool. The strategies proposed have been empirically evaluated with encouraging results. Future work involves addressing the problems posed to test case generation by the object-oriented paradigm, and will be mainly focused on the research of novel methodologies for input domain reduction and search space sampling.

7. REFERENCES

[1] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[2] B. Eckel. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002.

[3] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java. In *ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering*, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[4] S. Luke. ECJ 16: A Java Evolutionary Computation Library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2007.

[5] D. J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[6] J. C. B. Ribeiro, F. F. de Vega, and M. Z. Rela. Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing. In *SBRC WTF 2007: Proceedings of the 8th Workshop on Testing and Fault Tolerance at the 25th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 143–156. Brazilian Computer Society (SBC), 2007.

[7] J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega. eCrash: A Framework for Performing Evolutionary Testing on Third-Party Java Components. In *JAEM'07: Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas at the II Congreso Español de Informática*, pages 137–144, 2007.

[8] J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega. An Evolutionary Approach for Performing Structural Unit-Testing on Third-Party Object-Oriented Java Software. In *NICSO 2007: Proceedings of the 2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization (to appear)*, Studies in Computational Intelligence. Springer-Verlag, 11 2007.

[9] J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega. A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software. In *AST'08: Proceedings of the 3rd International Workshop on Automation of Software Test at the 30th International Conference on Software Engineering (to appear)*, 5 2008.

[10] J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega. Strongly-Typed Genetic Programming and Purity Analysis: Input Domain Reduction for Evolutionary Testing Problems. In *GECCO'08: Proceedings of the Genetic and Evolutionary Computation Conference (to appear)*, 7 2008.

[11] A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005.

[12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java Bytecode Optimization Framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[13] S. Wappler and J. Wegener. Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm. In *CEC'06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 851–858. IEEE, 2006.

[14] S. Wappler and J. Wegener. Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed gGnetic Programming. In *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1925–1932, New York, NY, USA, 2006. ACM Press.