Strongly-Typed Genetic Programming and Purity Analysis: Input Domain Reduction for Evolutionary Testing Problems

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

Mário Alberto Zenha-Rela
University of Coimbra
CISUC, DEI, 3030-290,
Coimbra, Portugal
mzrela@dei.uc.pt

Francisco Fernandéz de Vega
University of Extremadura
C/ Sta Teresa de Jornet, 38
Mérida, Spain
fcofdez@unex.es

ABSTRACT

Search-based test case generation for object-oriented software is hindered by the size of the search space, which encompasses the arguments to the implicit and explicit parameters of the test object's public methods. The performance of this type of search problems can be enhanced by the definition of adequate Input Domain Reduction strategies.

The focus of our on-going work is on employing evolutionary algorithms for generating test data for the structural unit-testing of Java programs. Test cases are represented and evolved using the Strongly-Typed Genetic Programming paradigm; Purity Analysis is particularly useful in this situation because it provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios.

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, Strongly-Typed Genetic Programming, Input Domain Reduction

1. INTRODUCTION

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* (ET) or *search-based test case generation* (SBTCG). Unit-test cases for object-oriented (OO) software consist of *method call sequences* (MCS), which define the test scenario. Each test case focuses on the execution of one particular public method – the *method under test* (MUT). One of the most pressing challenges faced by researchers in the ET area is the *state problem*, which occurs with objects that exhibit state-like qualities by storing information in private fields requiring all interaction to be performed through its public methods – a fundamental principle of OO programming known as *data encapsulation*.

The goal of the evolutionary search is to find MCS that define interesting state scenarios for the variables which will be passed, as arguments, in the call to the MUT. The input domain thus encompasses the *parameters* of the test object's public methods – including the implicit (i.e. the `this` object) and explicit parameters. Recent surveys indicate that *Input Domain Reduction* (IDR) strategies can greatly increase the performance of SBTCG problems [1].

2. PURITY ANALYSIS

Methods in OO languages often modify the objects that they access, including their parameters. However, some methods have no externally visible side effects when executed; these are called *pure methods*. A pure method is one which does not: perform input/output operations; write to any pre-existing objects; or invoke any impure methods [4]. More interestingly, important purity properties can be identified 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 SBTCG, 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 MCS that compose test cases. Also, it improves the potential quality of test cases by ensuring that no runtime exceptions are thrown by instructions that do not contribute to the definition of test scenarios.

3. TECHNICAL APPROACH

With our approach, test cases are represented and evolved using the Strongly-Typed Genetic Programming (STGP) paradigm. STGP was proposed with the intention of addressing the "closure" limitation of the Genetic Programming technique [2]. 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. 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. It is thus particularly suited for representing MCS for strongly-typed programming languages such as Java, as it enables the re-

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
ACM 978-1-60558-130-9/08/07.

```
Data: class under test
Result: purified function set
compute public methods list;
compute test cluster;
foreach public method do
    foreach parameter do
        annotate parameter purity;
    compute data types required table;
    compute data types provided table;
initialize EMCDG with data type nodes;
connect EMCDG nodes with call dependence edges;
remove irrelevant EMCDG edges;
create purified EMCDG;
create purified function set;
```

Figure 1: Input Domain Reduction algorithm.

duction of the search space to the set of *compilable* sequences by allowing the definition of constraints that eliminate invalid combinations of operations.

Our methodology involves encoding potential solutions (i.e. test cases) as STGP individuals, and MCS as STGP trees. Each tree subscribes to a Function Set, which is specified in correspondence to the constraints of the test cluster classes and defines the STGP nodes legally permitted in the tree. The Function Set contains the initially defined set of entries from which the STGP algorithm can choose when evolving test programs; our IDR strategy thus involves restricting the set of available functions to those that are effectively relevant to the search – i.e., those that are *impure* and thus contribute to the definition of state scenarios.

For modelling call dependences and defining the Function Set, an Extended Method Call Dependence Graph (EMCDG) [6] is employed. Our IDR strategy involves the removal of irrelevant edges from the EMCDG; this is performed by annotating the purity of parameters with the aid of the Soot Java Optimization Framework [5], and using this information to build the *purified EMCDG*. The *purified Function Set* is computed with basis on the purified EMCDG. The algorithm for the purified Function Set generation procedure is outlined in Figure 1.

4. EXPERIMENTAL STUDIES

Our IDR strategy was empirically evaluated with encouraging results. For the JDK 1.4.2 classes employed in the IDR experiment (Table 1), the statistics show a clear improvement; 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 IDR on the test case generation process was also visible on the case studies performed with the `Stack` and `BitSet` classes (Table 2) using our test case generation tool – *eCrash* [3]. For the `Stack` class, the number of generations required to attain full coverage (*gens*) using Purity Analysis was, on average, only 34% of that required without Purity Analysis. For the `BitSet` class the improvement is not as clear; still, the average percentage of test cases that accomplished full coverage within a maximum of 100 generations (*full*) increased approximately 6%.

Table 1: Input Domain Size Experiment Results.

	Function Set Entries	
	No Purity	**Purity**
`Stack`	12	7
`BitSet`	54	36
`BoolStack`	27	22
`ObjectVector`	43	28

Table 2: Test Case Generation Experiment Results.

	No Purity		**Purity**	
	gens	*full*	*gens*	*full*
`Stack`	4.4	80.0%	1.5	80.0%
`BitSet`	29.4	52.9%	30.9	58.7%

5. CONCLUSIONS

The search space of evolutionary testing problems can be significantly reduced by means of Purity Analysis. With our approach, test cases are represented using the STGP technique; Purity Analysis is particularly useful in this context, as it provides a means to automatically discard Function Set entries that do not contribute to the definition of interesting state scenarios, trimming down the set of method calls from which the algorithm can choose to those that are relevant to the test case generation process.

6. REFERENCES

[1] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164, New York, NY, USA, 2007. ACM Press.

[2] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[3] 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.

[4] 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.

[5] 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.

[6] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic 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.