The Journal of Systems and Software xxx (2014) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A Memetic Algorithm for whole test suite generation

Gordon Fraser[a,*], Andrea Arcuri[b], Phil McMinn[a]

[a] University of Sheffield, Department of Computer Science, 211 Regent Court, Portobello S1 4DP, Sheffield, United Kingdom
[b] Certus Software V&V Center, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

ARTICLE INFO

Article history:
Received 1 December 2013
Received in revised form 17 April 2014
Accepted 9 May 2014
Available online xxx

Keywords:
Search-based software engineering
Object-oriented
Evolutionary testing

ABSTRACT

The generation of unit-level test cases for structural code coverage is a task well-suited to Genetic Algorithms. Method call sequences must be created that construct objects, put them into the right state and then execute uncovered code. However, the generation of primitive values, such as integers and doubles, characters that appear in strings, and arrays of primitive values, are not so straightforward. Often, small local changes are required to drive the value toward the one needed to execute some target structure. However, global searches like Genetic Algorithms tend to make larger changes that are not concentrated on any particular aspect of a test case. In this paper, we extend the Genetic Algorithm behind the EvoSuite test generation tool into a Memetic Algorithm, by equipping it with several local search operators. These operators are designed to efficiently optimize primitive values and other aspects of a test suite that allow the search for test cases to function more effectively. We evaluate our operators using a rigorous experimental methodology on over 12,000 Java classes, comprising open source classes of various different kinds, including numerical applications and text processors. Our study shows that increases in branch coverage of up to 53% are possible for an individual class in practice.

1. Introduction

Search-based testing uses optimization techniques such as Genetic Algorithms to generate test cases. Traditionally, the technique has been applied to test inputs for procedural programs, such as those written in C (McMinn, 2004). More recently, the technique has been applied to the generation of unit test cases for object-oriented software (Fraser and Arcuri, 2013b). The problem of generating such test cases is much more complicated than for procedural code. To generate tests that cover all of the branches in a class, for example, the class must be instantiated, and a method call sequence may need to be generated to put the object into a certain state. These method calls may themselves require further objects as parameters, or primitive values such as integers and doubles, or strings, or arrays of values. The EvoSuite tool (Fraser and Arcuri, 2011) uses Genetic Algorithms to generate a whole test suite, composed of a number of test cases. Although empirical experiments have shown that it is practically usable on a wide range of programs (Fraser and Arcuri, 2012), Genetic Algorithms are a global search technique, which tend to induce macro-changes on the test suite. In order to cover certain branches, more focused changes are required. If, for example, somewhere in the test suite there is a particular integer variable, the probability of it being mutated during the search with a Genetic Algorithm is low, and so the optimization toward particular branches dependent on this value will take a long time. The difficulty of this problem becomes even more apparent when one takes into account string variables. Consider the example test case in Fig. 1: transformations of the string-based branching statement (see Section 3.3) provide guidance to the search for an input to reach the true-branch. However, even under very strong simplifications, a "basic" Genetic Algorithm would need an average of at least 768, 000 costly fitness evaluations (i.e., test executions) to cover the target branch. If the budget is limited, then the approach may fail to cover such goals. To overcome this problem, we extend the Genetic Algorithm used in the whole test suite generation approach to a Memetic Algorithm: at regular intervals, the search inspects the primitive variables and tries to apply local search to improve them. Although these extensions are intuitively useful and tempting, they add additional parameters to the already large parameter space. In fact, misusing these techniques can even lead to worse results, and so we conducted a detailed study to find the best parameter settings. In detail, the contributions of this paper are:

1 *Memetic Algorithm for test suite optimization*: We present a novel approach to integrate local search on test cases and primitive values in a global search for test suites.

* Corresponding author. Tel.: +44 114 22 21844
E-mail addresses: gordon.fraser@sheffield.ac.uk (G. Fraser), arcuri@simula.no (A. Arcuri), p.mcminn@sheffield.ac.uk (P. McMinn).

http://dx.doi.org/10.1016/j.jss.2014.05.032

```
class Foo {
  boolean bar(String s) {
    if(s.equals("bar"))
      // target
  }
}
```

```
Foo foo = new Foo();
String s = "test";
foo.bar(s);
```

Fig. 1. Example class and test case: in theory, four edits of s can lead to the target branch being covered. However, with a Genetic Algorithm where each statement of the test is mutated with a certain probability (e.g., 1/3 when there are three statements) one would have to be really lucky: if the test is part of a test suite (size 10) of a Genetic Algorithm (population 50) and we only assume a character range of 128, then even if we ignore all the complexities of Genetic Algorithms, we would still need on average at least $50 \times 4 \times 1/((1/10) \times (1/3) \times (1/128)) = 768,000$ fitness evaluations before covering the target branch.

2 *Local search for complex values*: We extend the notion of local search as commonly performed on numerical inputs to string inputs, arrays, and objects.
3 Test suite improvement: We define operators on test suites that allow test suites to improve themselves during phases of Lamarckian learning.
4 *Sensitivity analysis*: We have implemented the approach as an extension to the EvoSuite tool (Fraser and Arcuri, 2013b), and analyze the effects of the different parameters involved in the local search, and determine the best configuration.
5 *Empirical evaluation*: We evaluate our approach in detail on a set of 16 open source classes as well as two large benchmarks (comprising more than 12,000 classes), and compare the results to the standard search-based approach that does not include local search.

This paper is an extension of Fraser et al. (2013), and it is organized as follows: Section 2 presents relevant background to search-based testing, and the different types of search that may be applied, including local search and search using Genetic and Memetic Algorithms. Section 3 discusses the global search and fitness function applied to optimize test suites for classes toward high code coverage with EvoSuite. Section 4 discusses how to extend this approach with local operators designed to optimize primitive values such as integers and floating point values, strings and arrays. Section 5 then presents our experiments and discusses our findings, showing how our local search operators, incorporated into a Memetic Algorithm, result in higher code coverage. A discussion on the threats to validity of this study follows in Section 6. Finally, Section 7 concludes the paper.

2. Search-based test case generation

Search-based testing applies meta-heuristic search techniques to the task of test case generation (McMinn, 2004). In this section, we briefly review local and global search approaches to testing, and the combination of the two in the form of Memetic Algorithms.

2.1. Local search algorithms

With *local* search algorithms (Arcuri, 2009) one only considers the neighborhood of a candidate solution. For example, a hill climbing search is usually started with a random solution, of which all neighbors are evaluated with respect to their fitness for the search objective. The search then continues on either the first neighbor that has improved the fitness, or the best neighbor, and again considers its neighborhood. The search can easily get stuck in local optima, which are typically overcome by restarting the search with new random values, or with some other form of escape mechanism (e.g., by accepting a worse solution temporarily, as with simulated annealing). Different types of local search algorithms exist, including simulated annealing, tabu search, iterated local search and variable neighborhood search (see Gendreau and Potvin, 2010, for example, for further details). A popular version of local search often used in test data generation is Korel's Alternating Variable Method (Korel, 1990; Ferguson and Korel, 1996). The Alternating Variable Method (AVM) is a local search technique similar to hill climbing, and was introduced by Korel (1990). The AVM considers each input variable of an optimization function in isolation, and tries to optimize it locally. Initially, variables are set to random values. Then, the AVM starts with "exploratory moves" on the first variable. For example, in the case of an integer an exploratory move consists of adding a delta of +1 or −1. If the exploratory move was successful (i.e., the fitness improved), then the search accelerates movement in the direction of improvement with so-called "pattern moves". For example, in the case of an integer, the search would next try +2, then +4, etc. Once the next step of the pattern search does not improve the fitness any further, the search goes back to exploratory moves on this variable. If successful, pattern search is again applied in the direction of the exploratory move. Once no further improvement of the variable value is possible, the search moves on to the next variable. If no variable can be improved the search restarts at another randomly chosen location to overcome local optima.

2.2. Global search algorithms

In contrast to local search algorithms, global search algorithms try to overcome local optima in order to find more globally optimal solutions. Harman and McMinn (2010) recently determined that global search is more effective than local search, but less efficient, as it is more costly. With *evolutionary testing*, one of the most commonly applied global search algorithms is a *Genetic Algorithm* (GA). A GA tries to imitate the natural processes of evolution: an initial population of usually randomly produced candidate solutions is evolved using search operators that resemble natural processes. Selection of parents for reproduction is based on their fitness (survival of the fittest). Reproduction is performed using crossover and mutation with certain probabilities. With each iteration of the GA, the fitness of the population improves until either an optimal solution has been found, or some other stopping condition has been met (e.g., a maximum time limit or a certain number of fitness evaluations). In evolutionary testing, the population would for example consist of test cases, and the fitness estimates how close a candidate solution is to satisfying a coverage goal. The initial population is usually generated randomly, i.e., a fixed number of random input values is generated. The operators used in the evolution of this initial population depend on the chosen representation. A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for specific branches—to achieve branch coverage of a program—a common fitness function (McMinn, 2004) integrates the *approach level* (number of unsatisfied control dependencies) and the *branch*