Project of the Course: Hill Climbing Test Generator for Java

Paolo Tonella, Fondazione Bruno Kessler, Trento, Italy

Assignment

The goal of the project is to develop a test case generator for Java and to validate it empirically on the Java class `BinarySearch`. During Day 1, Day 2 and Day 3, students use Spoon to instrument the code and measure the fitness of a test execution against a coverage target. Then, they write a test generator that uses the fitness to guide the search for test cases that cover the coverage target. During Day 4, students apply the test generator to the Java class under test and collect metrics to answer the research questions of the empirical study. During Day 5, students present and discuss their results.

Day 1: Code Preprocessing

The goal of Day 1 is to familiarize with Spoon and to solve a simple code transformation exercise using Spoon. The code transformation aims at adding a public static field to the class under test. Such a transformation is a necessary prerequisite for the code instrumentation to be carried out on Day 2. Specifically, these are the tasks to be conducted:

1. Get familiar with Spoon (http://spoon.gforge.inria.fr). Include Spoon in your Java development environment by configuring Maven or Gradle as described in the home page, under menu item *Usage*. Try some of the examples available in the home page under *Getting started*, in particular those under *Transformation processor*, to understand which classes are available and what operations they provide. Get familiar with the Javadoc documentation (available under *Getting started*) of the main Spoon classes and interfaces.

2. Write the code to add a `public static` field of type `int`, named `fitness`, to the class under test `BinarySearch`. This can be achieved by means of the core factory provided by Spoon to create new fields and by adding the newly created field to the class under test, via method `addField` of class `CtClass`.

Day 2: Code Instrumentation

The goal of Day 2 is to instrument the code of the class under test so that upon execution it computes a fitness value that can be accessed by an external test case generator. For this project, the fitness value is simplified into an approximate measure of the approach level, given by the nesting depth of the target with respect to its parents. The tool has to first store the path from the target block of code to the method root into a list of AST nodes of type *code block*. Then, it iterates over such a list and it adds a statement that computes the value of `fitness` into each code block from the list. Specifically, these are the tasks to be conducted:

1. The statement to be processed has line of code equal to that of the current target (e.g., line 43 of `BinarySearch`). This can be checked via methods `getPosition` and `getLine`, applied to a `CtStatement` AST node. Processing of the target statement consists of the traversal of all parent AST nodes until the AST node representing the enclosing method is encountered. In such a traversal, all nodes of type `CtBlock` are collected and added to a list. The result will be a list of the code blocks from the target to the enclosing method.

2. The list of code blocks from the target to the enclosing method, collected in the previous step, is iterated and at the beginning of each code block a statement is added to compute the fitness value. The statement to be added has the form `fitness = Math.min(fitness, <`*depth*`>)`, where <*depth*> is a value equal to zero at the target statement and incremented at each iteration over the code blocks from the target to the enclosing method.

At this point, the instrumented code contains a public static field `fitness` that is computed upon execution of method `search` of class `BinarySearch`. It should be noticed that the initial value of `fitness` should be set externally (i.e., by the code generator) to a large `int` value at each new execution of method `search`, in order for the `min` operator to work properly.

Day 3: Test Generator

The goal of Day 3 is to write a hill climbing test generator for Java that takes advantage of the code instrumentation for fitness computation developed previously. The method under test is method `search` of class `BinarySearch`. This method has two inputs: an array of fixed size = 5 (parameter `array`) and the value to be searched (parameter `search`) in the array. Hence, we represent a solution to our test generation problem as an array of size = 6 components of `int` type. The first 5 values in the array represent the 5 components of the input parameter `array`; the 6th value represents the input parameter `search`. The hill climbing algorithm is used to evolve the candidate solution (initially generated randomly) so as to decrease its fitness value (in our problem instance, lower values of the fitness are better, since they indicate that the execution is closer to the target – indeed our fitness measures the distance from the target). The neighbourhood of the current solution consists of 6 alternative solutions, obtained by replacing each of the 6 components of the current solution with a new, randomly generated value. The new solution replaces the initial one only if its fitness is lower than the initial one. To measure the fitness, the test generator executes method `search` of class `BinarySearch` with the parameters extracted from the candidate solution and gets the value of field `fitness` from class `BinarySearch`. Students are required to implement also a random test case generator, to be used as a baseline for the empirical comparison. The random test case generator generates random solutions, i.e., it fills the solution array, of size 6, with 6 random numbers. Specifically, these are the tasks to be conducted:

1. Write a method to execute a candidate solution (an array of size 6) against the class under test. Such executor will invoke method `search` with the parameters extracted from the current solution and it will return the fitness value stored inside field `fitness` of the class under test. Before executing the method under test, the executor assigns a large value (e.g., 1,000) to the `static` field `fitness` of class `BinarySearch`, so as to make sure that the use of `Math.min` in the computation of the new fitness value (as done in the code instrumentation of Day 2) works properly.

2. Write a method to explore the neighbourhood of the current solution. In particular, 6 neighbouring solutions are obtained by replacing each of the six components of the current solution with a new random value (e.g., in the range [0 : 49]). Each neighbouring solution is executed against the class under test and the new solution replaces the initial one only if the fitness value decreases.

3. Write a method that implements the hill climbing algorithm. The current solution is initialized with 6 random values (e.g., in the range [0 : 49]). A new solution produced by the exploration of the neighbourhood is accepted only if it reduces the fitness. Test generation continues until a solution with fitness equal to zero is found or the maximum number of fitness evaluations is consumed (e.g., 300). In the former case, the solution is written into a JUnit test case.

4. Write a random test case generator that generates random values (still in the range [0 : 49]) until a solution with fitness equal to zero is found or the maximum number of fitness evaluations is consumed (e.g., 300).

For the purposes of statistical data collection (Day 4), both the hill climbing and the random test case generators should report either a 1 or a 0 to indicate if test case generation was successful (1) or not (0). They should also report the number of fitness evaluations consumed. Another useful functionality is an external loop that executes the test generator a large number of times (e.g., 1,000), collecting success flag and fitness evaluations consumed at each iteration, so that statistics can be later computed, such as average success rate and average fitness evaluations consumed.

Day 4: Empirical Data Collection and Analysis

The empirical study aims at answering the following research questions:

- **RQ1 (effectiveness)**: Is there any difference between the success rate of the hill climbing test generator and a random test generator?
- **RQ2 (efficiency)**: Is there any difference between the generation time (measured as number of fitness evaluations) taken by the hill climbing test generator and the time taken by a random test generator?

Students should collects the success flag indicator, 0 or 1, and the number of fitness evaluations consumed, over multiple (1,000) runs of the hill climbing test generator and of the random test generator. They should compute average success rate and average number of fitness evaluations consumed. The number of fitness evaluations across iterations can be displayed using boxplots (e.g., in the statistical environment R, a function `boxplot` is available). To determine if there is any statistically significant difference in success rate (effectiveness) or fitness evaluations (efficiency) between hill climbing and random, students can resort to the exact Fisher test for the success rate and to the Wilcoxon test for the fitness evaluations. All these statistical tests are available, e.g., in the statistical environment R. Additional plots, such as the plot of the fitness values over time, are also welcome.

Day 5: Presentation and Discussion of the Results

Students present their project using at most 10 slides (including the title slide; no animation allowed). They can describe their hill climbing test generator explaining the algorithm implemented and the technical solutions adopted. Then, they can present the empirical results obtained on Day 4 and their answers to the research questions. Finally, they can discuss their findings critically, considering the potential value of the test generator for developers, and they can present their ideas for possible future improvements of their test case generator. Any other research idea stimulated by the project is also welcome in the discussion.