TestFul: Automatic Unit-Test Generation for Java Classes*

Luciano Baresi and Matteo Miraz
Politecnico di Milano – Dipartimento di Elettronica e Informazione
Via Golgi, 42 – 20133, Milano (Italy)
(baresi|miraz)@elet.polimi.it

ABSTRACT

This paper presents *TestFul*, an Eclipse plugin for the generation of tests for Java classes. It is based on the idea of search-based testing, working both at class and method level. The former puts objects in useful states, used by the latter to exercise the uncovered parts of the class.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Test generation, Java classes, Search-based testing

1. INTRODUCTION

TestFul proposes an innovative approach for the (semi) automatic generation of unit tests for Java classes based on search-based approaches [8]. The generation of tests is seen as a search problem, and thus it is tackled through search-based algorithms (e.g., random, hill climbing, evolutionary computing). Initially these approaches were developed for procedural software, and only recently they have been adapted to cope with object-oriented systems [9, 13].

Object-oriented systems are particularly tedious when we want to test their features. One must put the objects in proper states and provide the correct values for the input parameters of the functions under test. For this reason, the tests for object-oriented systems are conceptually composed of two parts: the first creates the desired state of the object/system, while the second exercises the actual behavior.

Traditional search-based approaches do not take in account the internal states of objects, even if they explicitly target stateful systems. Reaching a proper configuration of the objects' state can be expensive, but we must consider that we do not need to create a new, dedicated state for each feature we want to exercise. The same state (configuration) can enable different features, and new interesting states can be reached from known ones. This means that a smart test generation solution must reuse known states for testing as many features as possible, and it must also exploit the newly obtained configurations to exercise other features.

These are the underpinnings of *TestFul* [10, 1]. It exploits the internal state of objects to drive the exploration of the search space (i.e., of all the possible tests). *TestFul* uses both statement and branch coverage on the class under test (CUT) and combines an evolutionary algorithm with a hill climbing to work respectively at class and method level. The former puts objects in useful states, used by the latter to exercise the uncovered parts of the class.

The main features of *TestFul* are:

- It proposes a **(semi-) automated** approach: the user is only asked for a few data used to augment the efficiency of the approach.
- It is **tailored to object-oriented systems**, since it capitalizes sequences of operations able to put objects in complex states. Instead, other works might waste some effort by applying the *divide et impera* paradigm and by focusing on different structural elements separately.
- It **employs a hybrid multi-objective evolutionary algorithm** to generate the best tests for a given class. Other works either do not use any kind of guidance (i.e., random testing), or they only use the information gathered from the control-flow graph.

The first experiments confirm the validity of the approach. We compared *TestFul* with state of the art approaches on an independent benchmark, and our solution was able to generate tests able to reach higher statement and branch coverages than the others.

This paper presents the first release of the tool called *TestFul*, which is an Eclipse plug-in available as an open source project at `http://code.google.com/p/testful`.

2. KEY PRINCIPLES

At the beginning, *TestFul* analyzes the class under test to figure out all the classes that might be involved in the test (*test cluster*). This is done by considering the CUT and by transitively including the types of all parameters of all public methods and constructors. Since abstract classes

*This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977 SM-Scom.

ICSE '10 May 2-8 2010, Cape Town, South Africa

and interfaces can be used as formal parameters, *TestFul* also considers the additional classes specified by the user.

For each type contained in the test cluster, *TestFul* creates a set of variables. To enable polymorphic behavior, it stores an object of type `A` either in a variable of the same type or in a variable whose type is an ancestor of `A` (i.e., `A`'s superclasses or interfaces implemented by `A`). Conversely, when an object is selected from a variable of type `A`, it may be an instance of `A` or of one of its sub-classes.

TestFul renders any test for the CUT by using a sequence of operations on those variables. Each test starts from a clean environment, in which all variables are not initialized; each operation can both use the variables as actual parameter and store the result in a variable. A test is rendered by using three kinds of operations:

- **assignment** assigns a primitive value —i.e., boolean, byte, integer, long, float, double— to a variable.
- **object creation** creates an object by calling one of the constructors, using variables as actual parameters, and stores the created object in a variable.
- **method invocation** invokes a method, using variables as receiving object and actual parameters. If the method returns a non-void value, it is stored in a variable (or it is discarded). Note that if the method mutates the state of some objects, these changes affect subsequent operations.

Through this structure, *TestFul* randomly generates some tests, which are then evolved using search-based techniques towards the optimal ones. For this purpose, *TestFul* works both at class and method level, and uses respectively an evolutionary algorithm and a hill climb.

The former focuses on the whole class, seeking for a test able to put objects in interesting states. To recognize these states, we use the level of structural coverage (both statement and branch coverage) achieved by the test as heuristic. The higher it is, the better the test is able to exercise the class, and put objects in interesting states. The evolutionary algorithm uses this information as guidance to drive the recombination of tests (i.e., their sequences of operations).

The latter drills down through the conditions contained in the different methods. Its goal is to cover *branches* never exercised before and test features not executed yet. We only consider conditional statements already exercised, and we target one of the branches never taken. The condition associated with the selected branch is already evaluated by the original test, hence we let our approach to save on effort.

The original test may contain operations that exercise features that have no impact on the selected branch. To converge faster, we reduce the test by pruning these operations. This is achieved by using information regarding the *type* of each method, its violations of the *information hiding* principle, and its side-effects.

As for types, a method can be: a *mutator*, when it may change the object's state (this is the default value), a *worker*, when it does not change the state, but it may perform some additional computations, an *observer*, when it does not change the state and does not perform any additional computation, or a *static*, when it does not belong to any object.

A method can violate the *information hiding* principle by exposing parts of its internal state, as done in the first two methods of the class of Figure 1. In this example, the user should specify that method `getState` exposes the object's state and also that the parameter of `setState` becomes part of the state.

```
class NoHiding {
  private List state;
  public void setState(List list) {
    this.state = list;
  }
  public List getState() {
    return this.state;
  }
  public static copy(List from, List to) {
    to.addAll(from);
  }
}
```

Figure 1: A class with particular methods.

Moreover, a method can embed *side-effects* on its parameters by mutating their state or by exchanging part of them (e.g., method `copy` of Figure 1). The user can specify them by using the wizard to set the properties of the method's parameters.

Note that this information is not used directly during the test generation process. Moreover, if that information is missing or incorrect, the quality of the result is preserved, but *TestFul* may take longer to generate good tests.

The search process randomly changes the test, executes it, monitors the condition that controls the selected branch, and calculates:

$$distance(a \oplus b) := \begin{cases} +\infty & \leftrightarrow \quad condition\ not\ executed \\ -\infty & \leftrightarrow \quad target\ branch\ executed \\ |a-b| & \leftrightarrow \quad otherwise \end{cases}$$

Where $a \oplus b$ is the selected condition, a and b are constants, local variables, or field variables, and $\oplus$ is an admissible relational operator. This distance (its minimum value, if the condition is executed multiple times) is used as guidance for the search algorithm.

If the search process is successful, the generated test has an improved structural coverage. The evolutionary algorithm that works at class level recognizes this and uses it as new starting point to reach even higher structural coverages.

2.1 Implementation

TestFul is fully implemented in Java. Its internal architecture is modular, to allow one to easily integrate new coverage criteria, and organized around three modules: the instrumenter, the test generator, and the Eclipse GUI.

The **instrumenter** exploits *SOOT* [12] to statically analyze the class under test and insert the tracking code used to measure statement coverage, branch coverage, and the distance between the actual values used to evaluate the condition and those needed to reach a branch.

The **test generator** implements the *TestFul* approach. It is based on *jMetal*'s evolutionary algorithm [6]. It uses the Java class-loading facility to load the instrumented classes and ensure isolation among concurrent evaluation of tests.

The **Eclipse GUI** integrates *TestFul* with the Eclipse Java Development Tool to provide users with a standard development environment, and improve their user experience.

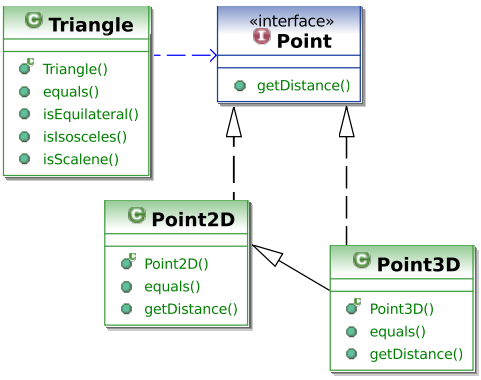

Figure 2: Class diagram of the use case.

3. EXAMPLE

To exemplify how the tool (approach) works, we consider a class `Triangle` (Figure 2) that renders triangles in an euclidean space. Even if this is a simple class, and the tool accepts more complex ones, it owns all the elements to explain the key features of *TestFul*.

The class' constructor wants three `Points` as parameters, that is, the three vertexes of the to-be-created triangle, and throws an exception if one of the parameters is null or they do not satisfy the triangular inequality. The class also provides methods to check whether a given triangle is equilateral, isosceles, or scalene, and to compare two triangles.

The class uses interface `Point` to render its vertexes; this interface offers a method to calculate the distance between two points. Classes `Point2D` and `Point3D` implement the interface in a two- and three-dimensional space, respectively.

To create a test for the class, we exploit the Eclipse GUI and right-click on the class under test (CUT) in the package view. The first step of the wizard (Figure 3) shows on the left-hand side all the classes belonging to the test cluster. *TestFul* checks the validity of the test cluster, that is, whether if it can create an instance of every class in the list to used as formal parameter of methods or constructors. In the example, the constructor of `Triangle` depends on interface `Point`, but *TestFul* is not able to create any instance of that type. For this reason, the wizard reports an error and asks the user for concrete implementations of the interface `draw.Point`. Consequently, we can add both `draw.Point2D` and `draw.Point3D` to check whether one can mix different coordinate systems in a single triangle. Moreover, the users can inspect each class and modify the properties associated with its methods and constructors. They can set the type of the method, its violation of the information hiding principle, and its side-effects on parameters.

As soon as the specification step is complete, and sufficient instances can be created, the tool instruments the classes to insert some tracking code. This will allow *TestFul* to gather information about the behavior of the class under test (e.g., to measure branch coverage).

Finally, the users can set some constraints on the generation process. Default values are always feasible and safe, but for example the actual generation may take too long. This is why, given the simplicity of class `Triangle`, we decided to allow the tool to run for a couple of minutes.

The generation is done in a background process, and does not forbid the users to work on the other classes of the

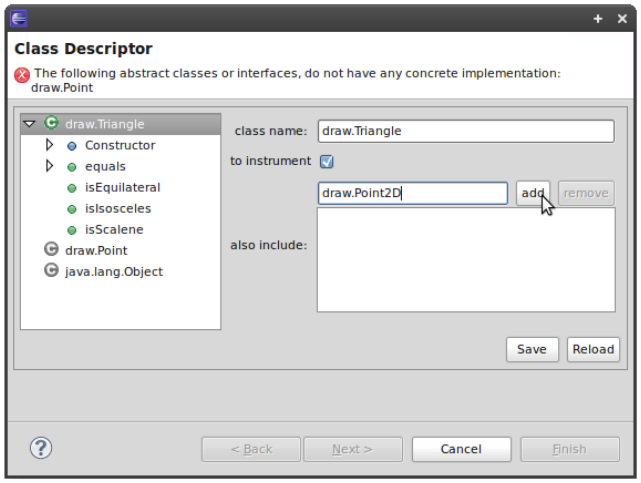

Figure 3: *TestFul* wizard.

project in the meanwhile. Once the tests are ready, they are saved in a project's sub-directory as jUnit tests (a test fragment is reported in Figure 4). Users can review them manually or through some automatic tools, or they can use them directly to ensure that newer versions of the class comply with the current implementation (i.e., to perform regression testing).

```
i_1 = 6;
i_3 = -1356361227;
p_3 = new draw.Point3D(i_1, i_1, i_3);
p3d_2 = new draw.Point3D();
assertEquals(1356361227.0,
             p3d_2.getDistance(p_3));

p_2 = null;
try {
  t_0 = new Triangle(p_1, p_2, p_3);
  fail("Expecting a java.lang.Exception");
} catch (java.lang.Exception e) {
  assertEquals("The point cannot be null",
               e.getMessage());
}
```

Figure 4: Extract of the generated test.

4. RELATED WORK

As for search-based techniques for the generation of functional tests, we must start with *random testing* [4, 5], which is probably the most famous approach. It employs a very simple technique since it performs a random sequence of invocations on the system under test. Notwithstanding its simplicity, random testing can be as effective as other traditional approaches [3]. When failures are detected, random testing tools are able to provide witnesses, which are sequences of operations able to reveal the failure.

To augment effectiveness, some works [9] also propose *adaptive random testing* (ART) to ensure that generated values are equally distributed over the input domain [2]. The idea is that the more distant those values are, the better they

reveal failures. However, empirical studies show that ART tools do not discover failures earlier; instead, they reveal a different set of failures.

Other approaches, such as [11], enhance traditional random testing with taboo-search or guided generation techniques since the search process is directed towards the satisfaction of a goal.

Focusing on structural testing, the goal is to maximize the coverage for a given criterion (e.g., all branches in the system). McMinn [8] detects two ways to tackle the problem: coverage-oriented and structure-oriented approaches. The former rewards the tests that cover more structural elements. For example, Watkins [14] tries to achieve full path coverage on stateless systems, but his solution is not able to provide enough guidance to the search process, and thus it is not able to achieve good results. The latter tackles each uncovered structural element identified by the coverage criterion separately [7]. Before applying the search algorithm, the system under test is analyzed to select the set of structural elements of interest. Then, the algorithm selects one of these structural elements and uses a search algorithm to generate a test able to reach it.

These proposals work on stateless systems: they consider a single function invocation and generate the input parameters to reach the selected structural element. In contrast, Tonella [13] focuses on object-oriented systems, and for each branch of the class under test, he searches for a sequence of operations able to prepare the state of objects and exercise the selected branch. His work does not capitalize the state of objects: the fitness function is the same as that used by the approaches that operate on stateless systems, and when a new branch is targeted, the search process re-starts from scratch.

We also compared *TestFul* against some of these approaches able to work on stateful systems [1], namely *jAutoTest* (a version of AutoTest [9], developed for Java), *randoop* [11], and *etoc* [13]. Figure 5 summarizes the main results, measured in terms of branch coverage. Even with a limited amount of time, *TestFul* outperforms the other approaches and generates tests with higher branch coverage. Note that *TestFul* would generate even better tests if it were run longer.

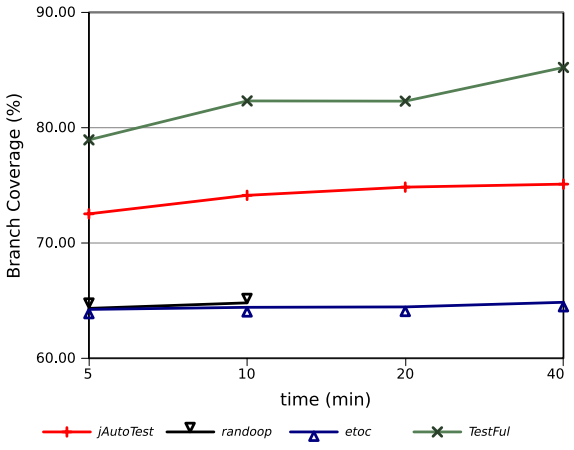

Figure 5: *TestFul* vs. the state of the art.

5. CONCLUSIONS

This paper introduces *TestFul*, a search-based framework for the automatic test generation for Java classes. *TestFul* generates tests by working at two levels. The first level is designed to recognize and reuse useful state configurations. The second level exploits these objects' states to reach uncovered branches, which in turn may put objects in new states. Despite its efficacy, *TestFul* is simple and effective. It proficient use does not require particular training.

Acknowledgments

The authors want to thank Pier Luca Lanzi for the many fruitful discussions on the underpinnings of *TestFul*.

6. REFERENCES

[1] L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: an Evolutionary Test Approach for Java. In *Proceedings of the 3rd International Confenrence on Software Testing, Verification and Validation (ICST)*, 2010.

[2] T. Chen, H. Leung, and I. Mak. Adaptive Random Testing. In *Proceedings of Advances in Computer Science*, volume 3321/2005, pages 320–329, 2004.

[3] T. Y. Chen and Y.-T. Yu. On the Expected Number of Failures Detected by Subdomain Testing and Random Testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, 1996.

[4] D. Hamlet. When Only Random Testing Will Do. In *Proceedings of the 1st International Workshop on Random Testing*, pages 1–9, 2006.

[5] R. Hamlet. Random Testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[6] jMetal: A Framework for Multi-Objective Optimization. `http://jmetal.sourceforge.net/`.

[7] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[8] P. McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[9] B. Meyer, I. Ciupa, A. Leitner, and L. Liu. Automatic Testing of Object-Oriented Software. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129. Springer, 2007.

[10] M. Miraz, P. L. Lanzi, and L. Baresi. TestFul: Using a Hybrid Evolutionary Algorithm for Testing Stateful Systems. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, pages 1947–1948, 2009.

[11] C. Pacheco, S. K. Lahiri, and T. Ball. Finding Errors in .NET With Feedback-directed Random Testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96, 2008.

[12] SOOT: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

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

[14] A. Watkins. The Automatic Generation of Test Data Using Genetic Algorithms. In *Proceedings of the 4th Software Quality Converence*, pages 300–309, 1995.