Summary on the Human Competitiveness of *TestFul*

Matteo Miraz
Dipartimento di Elettronica e Informazione, Politecnico di Milano
piazza Leonardo da Vinci, 32
20133 - Milano (Italy)
email: `miraz@elet.polimi.it`
phone: +39 02 2399 3770

June 6, 2011

1 Introduction

Software testing should be a fundamental activity of any software development process: even if a systematic testing activity does not guarantee error-freeness, a good and sound campaign would increase the quality of the product. Tassey[9] estimated in $20 billion the amount of money that could be saved every year if better testing were performed.

Beizer[3] estimated that an adequate testing campaign might require up to half of the software development effort. To alleviate this burden, the research community have proposed several ways to (semi-)automatically generate test data. These approaches rely on deterministic techniques, well-established in the Software-Engineering world (e.g., SAT solvers). However, the proposals presented so far "face challenges in generating test inputs to achieve high structural coverage when they are applied on complex programs in practice"[11].

To move beyond these problems, we employ an Evolutionary Algorithm and we propose *TestFul*. It uses a *holistic* approach to make the state of object evolve, to enable all the features the class provides, and to generate the shortest test with the utmost coverage for the class under test. We employ several complementary coverage criteria to drive the evolutionary search. We aim to generate tests with high fault detection effectiveness. To this end, we consider the system from complementary perspectives and we combine white-box analysis techniques with black-box ones. The evolutionary search is completed with a local one, and we establish a synergic cooperation between them. The evolutionary search concentrates on evolving the state of objects, while the local search detects the functionality not yet exercised, and directly targets them.

The results we achieved so far exceeded our expectations. Since the early stages of *TestFul*, we were noticed how it outperforms other automated ap-

proaches [6]. Recently, we shift our focus on the human-competitiveness of our approach. This paper briefly summarizes the main results we achieved so far. Section 2 considers a controlled experiments, where students were asked to generate tests for non-trivial classes. Section 3 instead presents a error in a widespread library (the standard Java collections) which other automated approaches does not find (but it is present in the official bug database).

2 Controlled Experiment

To investigate the effectiveness in detecting faults, that is the quality, of generated tests, we employed mutation analysis [4, 1]. Mutant operators generate multiple mutated version of the code of interest, each one with a single fault seeded. Tests are judged according to the number of mutated versions they are able to detect (*kill*). However, the application of mutant operators might create *equivalent mutants*, which are semantically identical to the original program, hence tests cannot kill them. For example, $a > b$ and $a \geq b$ are equivalent if a cannot be equal to b. To correctly judge the fault-detection ability of tests, one should (manually) identify and prune equivalent mutants.

Although mutation analysis requires huge computational effort, it provides a good estimation of the fault-detection abilities of tests [1], allowing one to compare different ways to generate them.

In particular, Mouchawrab et al. [7] compare the fault detection effectiveness of state testing against structural testing on classes with state-driven behavior. They performed a series of controlled experiments involving students from two universities: Carleton university (Canada) and Università del Sannio (Italy). One of the three projects[1] they use as benchmark is `OrdSet`, which manages a bounded and ordered set of integers, and provides operations to add and remove an element, and merge two sets. Even if this project comprises a single class with a limited number of lines of code, its complexity is comparable to the other two projects of their study.

Since the source code of class `OrdSet` and the mutants they used are publicly available —through the Software Infrastructure Repository [5]— and [7] provides sufficient data, we replicated the statistical study. However the authors did not provide the tests they used in their experiments (because of copyright issues) and thus we could only perform a comparison on the overall mutation score, while we could not compare the *non-equivalent* mutation score or verify whether the student-written tests and those generated by *TestFul* are complementary.

Mouchawrab et al. used `MuJava` to automatically generate a large set of mutants (more than 800), but they did not remove or mark the equivalent ones, hence we performed this task manually. Table 1 reports the average mutation score and the standard deviation of the tests reported in [7][2] and those generated

[1] The other two projects used in [7] manage multi-threaded applications, which are currently not supported by *TestFul*.

[2] For the sake of simplicity, we use the same notation used in [7], and we label with "code"

Table 1: Mutation score and t-tests results.

Provenience	***all mutants*** **mean**	**(std)**	***p-value***
Carleton Code	56.15%	(19.99%)	0
Carleton State	50.27%	(17.20%)	0
Sannio Code	70.31%	(12.69%)	0.0004
Sannio State	71.96%	(12.41%)	0.0004
TestFul "du"	89.32%	(0.18%)	—

by the three configurations of *TestFul*. With these last tests, we were also able to prune equivalent mutants and thus we also report the mutation score for non-equivalent mutants.

According to these data, *TestFul* outperforms the tests generated by the students, since its tests have both a higher average mutation score and a lower standard deviation. To have a statistical confirmation of this conjecture, we also performed a statistical hypothesis test between H_0 : "*There is no statistical difference between the quality of the tests created by* TestFul *and those generated by human beings*" vs H_1 : "*There is statistical difference between the quality of the tests created by* TestFul *and those generated by human beings*", and the p-values are reported in Table 1.

All the p-values are close to 0, hence we can safely reject the null hypothesis and state that tests generated by *TestFul* are better than those created by human beings.

Finally, as for costs, the tests generated by *TestFul* are (obviously) cheaper: its generation is completely automatic and it only requires 30 minutes (instead of the three or four hours given to the students). The cost for executing each generated test is also reasonable with an average of 7ms.

2.1 Replicated Experiment

In order to gain higher confidence on the human-competitiveness of *TestFul*, we performed a controlled experiment similar to the one presented in the previous section. In particular, we focused on the students attending the "Software Engineering" course, which is taught on the last year of our Bachelor degree. The course focuses on the fundamentals of software engineering, including object-oriented programming and testing. The students were extremely motivated to perform well in the experiments, because their final marks in the course also depend on their performance in the experiment.

The results we achieve were similar to those achieved in the previous experiment, and they are reported in Figure 1 and Table 2. In particular the latter reports the p-value of the Wilcoxon test between *TestFul* and random-testing and the test generated by the students. Again, there is statistical evidence that *TestFul* generate tests with a higher quality than the students. Note that the

(C) the *structural tests*, and with "state" (S) the *state tests*.

random generation is one of the best available techniques to generate tests [2]. In line with our previous results [6], *TestFul* outperforms the random test generation.

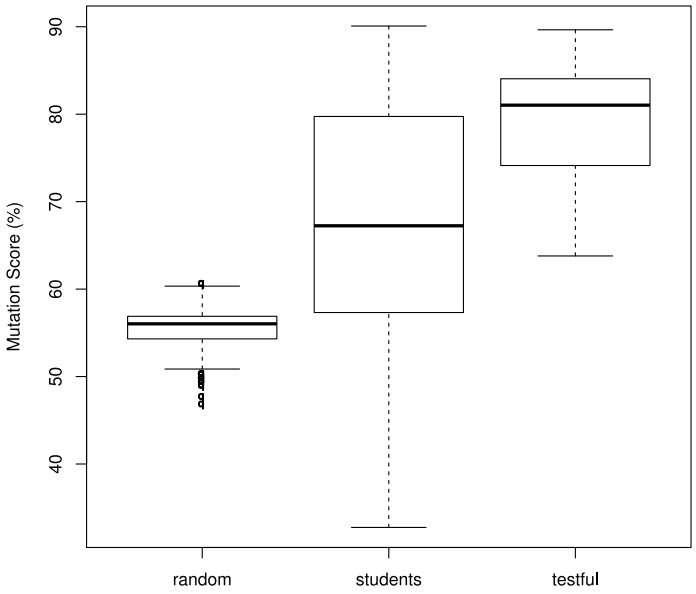

Figure 1: Mutation Score

Table 2: Mutation score and t-tests results.

Provenience	***Samples***	***all mutants*** **mean**	**(std)**	***p-value***
random	984	55.44%	(2.22%)	$<$ 2.2e-16
Students	37	68.25%	(13.54%)	2.26e-07
TestFul "du"	1000	79.31%	(5.71%)	—

3 Real Fault

We run TestFul on some classes of the Sun's Java Collection Framework, and we found a bug which is reported in the sun's official bug database. Interestingly, other automated approaches and sun's paid programmers were not able to find such error, albeit "*container classes are the de facto benchmark for testing software with internal state*" [2].

Collections in package `java.util` are able to handle self-referring collections:

```
Collection<Object> c = new ArrayList<Object>();
c.add(c);
```

```
System.out.println(c.toString());
// prints "[(this Collection)]"
```

However, they are not able to deal with two collections that refer each other reciprocally:

```
Collection<Object> a = new ArrayList<Object>();
Collection<Object> b = new ArrayList<Object>();
a.add(b);
b.add(a);
System.out.println(a.toString());
```

In this case, method `toString` enters an infinite recursive loop terminated by the virtual machine by throwing a `StackOverflowError`. *TestFul* reported such an error automatically, while several works on the automatic generation of tests [10, 12, 8] targeted this package, but they did not discover it. The official Java bug database reports the error as open issue (entry 4275605).

References

[1] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE TSE*, 32(8):608–624, 2006.

[2] Andrea Arcuri. Longer is Better: On the Role of Test Sequence Length in Software Testing. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[3] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the praticing programmer. *IEEE Computer*, 11(4):34–41, Apr 1978.

[5] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[6] Matteo Miraz, Luciano Baresi, and Pier Luca Lanzi. TestFul: an Evolutionary Test Approach for Java. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[7] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[8] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[9] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology RTI Project, 2002.

[10] Paolo Tonella. Evolutionary Testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[11] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 611–620, New York, NY, USA, 2011. ACM.

[12] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, 2005.