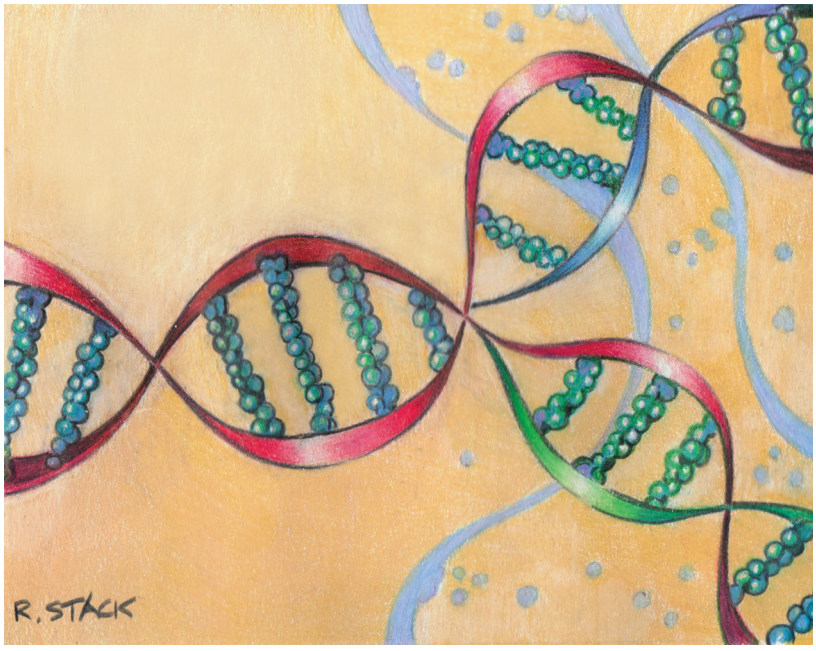
Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Mutation Testing

Pedro Reales, Macario Polo, José Luis Fernández-Alemán, Ambrosio Toval, and Mario Piattini

Testing is expensive for two reasons: inefficient test suites and ineffective defect detection. Since its inception, mutation testing has proven to be a highly cost-effective test technique by improving the hit rate of test suites and by creating test cases where absolutely necessary. The authors introduce mutation testing and provide an overview of current hot technologies. I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —*Christof Ebert*



SUPPOSE YOU MUST contract a copyeditor or proofreader for an editorial project. To select the best one, you could provide the candidates with deliberately misspelled text and ask them to find all the inserted typos and errors—Figure 1 shows a good example. With this process, you're achieving a double goal: evaluating the quality of the candidates and finding someone who can make sure your own text is free of typos. Of course, some errors can pass unnoticed by even the best reviewer, such as the change of "that is" to "that's" (first line), which strictly speaking, isn't a fault.

The idea behind mutation testing is exactly the same: we generate mutants from a program (the program under test, also called the *original program* in the mutation context) that we suppose is correct. Each mutant holds at least one artificial change. Usually, the change is a fault that can be found with a good test case, but sometimes the change is a code optimization or deoptimization that will lead the mutant to exhibit the same behavior, such as the "that's" change in the first line of Figure 1: these are *equivalent mutants*. So, mutation testing achieves two goals as well[1]: evaluating test suite quality (the more artificial faults a test suite detects, the higher its quality), and once we have a good test suite, executing its test cases against the original program to find errors.

0740-7459/14/$31.00 © 2014 IEEE

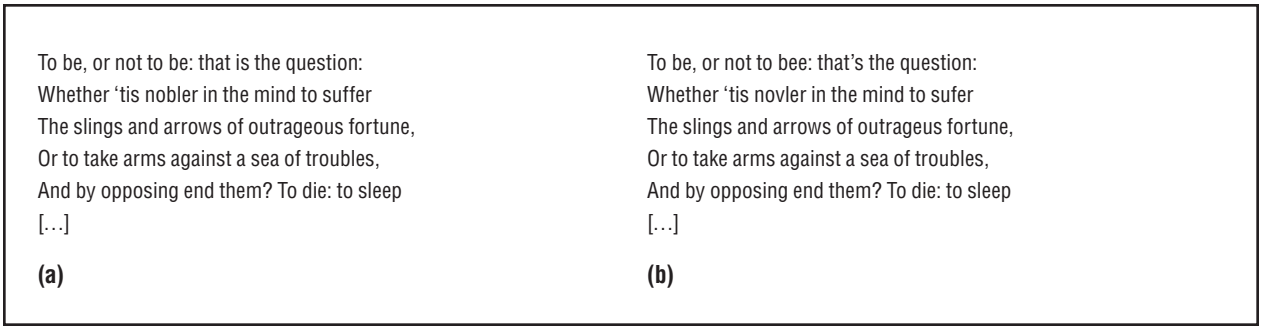


FIGURE 1. Spot the errors: (a) the real fragment of *Hamlet* and (b) a mutated version.

Mutation testing, proposed in 1978 by Richard A. DeMillo and his colleagues[2] is an effective technique: if a test suite finds all the artificial errors inserted in the mutants and finds no fault in the original, it's likely that the program under test is free of them. Obviously, the validity of this affirmation depends on the nature of the artificial fault: some of them are better than others. This testing technique has been used in the research arena to check the effectiveness of new proposed testing techniques, but it hasn't been used until recently in industry due to its costs and the lack of knowledge and industrial tools. In this article, we give a short overview of the main characteristics of mutation tools.

Nuts and Bolts

Mutation testing lets testers identify artificial faults inserted in copies of the system under test (SUT). There are three main steps to performing a mutation analysis:

- *Generate mutants.* A mutant is a copy of the original system with a small syntactic change that can suppose a fault. Syntactic changes are created via mutation operators, well-formed rules that determine the changes. Mutants are generated by automatic tools that apply a set of operators to each sentence of the SUT, thus producing a huge number of mutants.
- *Execute the original system and mutants.* The designed test cases must be executed against the original system and the mutants to compare each output's mutant with the original output. When a mutant's output differs, it's said that the mutant is "killed" by the test case that produced that output.
- *Result analysis.* The mutation score is a number between 0 (0 percent, low quality) and 1 (100 percent, high quality) that determines test quality. To calculate the mutation score, we need to identify the equivalent mutants.

These three steps are costly[1] in terms of storage requirements (to store all the mutants), high computational requirements (to create mutants and execute test cases against the original and the mutants), and human efforts (to identify equivalent mutants to calculate the mutation score). Researchers have developed different techniques to reduce these costs,[1] which Table 1 summarizes. We can classify them into three groups: mutant-reduction techniques, which reduce the total number of mutants with a small loss of effectiveness; execution kind, which are advanced computer architectures or new mutant generation and execution techniques; and mutation type, which helps analyze mutant and original outputs in different ways to kill mutants. This last group is especially helpful in reducing the cost of testing tasks.

Due to the cost and the nature of mutation tasks, it's important to have automatic tools that can perform mutation analyses (generating mutants, executing test cases, and calculating the mutation score automatically). Likewise, it's important that mutation tools include as many cost-reduction techniques as possible to reduce storage and computational costs.

Mutation Tools

There exist many mutation tools for different languages and technologies. Because the use of tools is essential for the actual adoption of mutation testing, Table 2 lists some tools for Java, C#, and C/C++. Besides programming language, the set of implemented mutation operators is an essential factor in determining the corresponding application fields.

Cost-Reduction Techniques

The set of techniques included in a mutation tool is a very important

TABLE 1

Mutation cost reduction techniques.*

Nature	Category	Technique	Description
Basic techniques (don't reduce mutation costs, but are used as base techniques)	G&E	Compiler-based	Source code is mutated and compiled for each mutant
	Mutation type	Strong mutation	The mutated sentence must be reached, and an erroneous state must be produced and propagated through the program output, which the test case observes
		Functional qualification	Differences in program states aren't compared; the fault is found when the oracle verdict (pass/fail) differs in the program under test and in the mutants
Cost-reduction techniques	Mutant reduction	Selective mutation	Use of a reduced set of operators
		Mutant sampling	Use of a random subset of the whole mutant set
		Higher-order mutation	Insertion of more than one fault in each mutant
		Mutant clustering	Selection of a subset of mutants based on a clustering process
	G&E	Compiler-integration	Mutants are directly generated by an instrumented compiler
		Mutant schemata	The original program and the mutants are combined into a single file; the version to be executed is selected at runtime
		Byte-code translation	Faults are inserted in the compiled code, not in the source
		Parallel execution	Test cases are executed in parallel in several machines
	Mutation type	Weak mutation	The mutated sentence must be reached and an erroneous state must be produced; the program execution is stopped after the mutated instruction and the possible state differences are observed
		Flexible weak mutation	The state of the whole system is compared on different locations during its execution

** G&E = generation and execution.*

characteristic that can determine a tool's suitability for our testing project. Table 3 shows the mutation techniques included in each tool described in the previous section.

Generally speaking, each tool only implements one or two techniques. Table 3 shows that selective mutation has been widely used. Mutant schemata,[3] byte-code translations,[4] and weak mutation[5] have been implemented in at least three tools. From the tools listed in Table 3, Bacterio includes these three techniques, Javalanche and muJava two of them, and Judy, Jumble, Milu, MuClipse, Mugamma, Muformat, and Testooj only implement one of them. Higher-order mutation is a promising technique[6] that has emerged in the last generation of mutation tools (Milu, Testooj, and Bacterio). The fact that the other techniques have been implemented in just a few tools indicates the implementation costs of these techniques.

A good mutation-testing tool should include as many cost-reduction techniques as possible. If you're planning to use mutation, we offer the following tips:

- Select a tool with the appropriate operators for your test requirements.
- Try to apply as many cost-reduction techniques as possible, taking into account that some of them lose effectiveness.
- If your system is too large, create mutants only for its most critical parts and use another coverage criterion for the others.
- Create re-executable tests compatible with your testing tools

TABLE 2

Java, C#, and C/C++ mutation tools.

Name	URL	Language	Application field	Mutation operators
AjMutator	www.irisa.fr/triskell/Softwares/protos/AjMutator	Java	Aspect Java programs	Aspects
Bacterio	www.alarcosqualitycenter.com/index.php/productos/bacterio	Java	Java complex applications	Traditional and interface
Certitude	www.springsoft.com/products/functional-qualification/certitude	C++	Hardware verification in microelectronics industry	Traditional
Cream	http://galera.ii.pw.edu.pl/~adr/CREAM/	C#	Dependability of computer systems	Class
Csaw	http://computing.open.ac.uk	C	Real-time embedded systems	Traditional
ESTP	www.uic.edu.hk/~XinFeng	C	Investigating testing strategies	Traditional
ExMan	http://faculty.uoit.ca/bradbury/conman	Java	Concurrent programs	Concurrency
Insure++	www.parasoft.com/products/insure/index.htm	C/C++	Runtime analysis and memory error detection	Traditional
Javalanche	www.javalanche.org	Java	Java complex applications, industry	Traditional
JavaMut	www.laas.fr/2-27719-Home.php	Java	Java programs, research	Traditional and class
Judy	http://madeyski.e-informatyka.pl	Java	Java complex applications, industry	Traditional and class
Jumble	http://jumble.sourceforge.net	Java	Java programs, mostly for research	Dynamic
Milu	www.cs.ucl.ac.uk/staff/Y.Jia/Milu/	C	Investigating both first-order and higher-order mutation testing	Traditional
MuClipse	http://muclipse.sourceforge.net	Java	Eclipse plugin for Java programs based on muJava, mostly for research	Traditional and class
Muformat	http://research.cs.queensu.ca/~mzulker	C	Identification of software vulnerabilities	Traditional
Mugamma	www.cc.gatech.edu/~harrold	Java	Java programs, mostly for research	Traditional and class
muJava	http://cs.gmu.edu/~offutt/mujava	Java	Java programs, mostly for research	Traditional and class
Proteum	www.icmc.usp.br/~delamaro	C	Investigating incremental testing strategies	Traditional and interface
Sesame	http://homepages.laas.fr	C	Fault-tolerant planning in robots	Dynamic
Testooj	http://alarcos.inf-cr.uclm.es/testooj3	Java	Java programs, mostly for research	Traditional and class

TABLE 3

Cost reduction techniques implemented in each tool.*

Tools	Basic techniques: G&E: Compiler-based	Basic techniques: Mutation type: Strong mutation	Basic techniques: Mutation type: Functional qualification	Cost reduction techniques: Mutant reduction: Mutant sampling	Mutant reduction: Selective mutation	Mutant reduction: Higher-order mutation	Mutant reduction: Mutant clustering	Cost reduction techniques: G&E: Compiler-integration	G&E: Mutant schemata	G&E: Byte-code translation	G&E: Parallel execution	Cost reduction techniques: Mutation type: Weak mutation	Mutation type: Flexible weak mutation
AjMutator	Yes	No	Yes	No	Yes	No	No	No	No	No	No	No	No
Bacterio	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes
Certitude	Yes	No	Yes	No	Yes	No	No	No	No	No	No	No	No
Cream	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Csaw	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
ESTP	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	no
ExMan	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Insure++	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Javalanche	No	No	Yes	No	Yes	No	No	No	Yes	Yes	Yes	No	No
JavaMut	Yes	Yes	No	No	Yes	No	No	Yes	No	No	No	No	No
Judy	Yes	No	Yes	No	Yes	No	No	No	Yes	No	No	No	No
Jumble	No	No	Yes	No	Yes	No	No	No	No	Yes	No	No	No
Milu	Yes	Yes	No	No	Yes	Yes	No	No	No	No	No	No	No
MuClipse	Yes	No	Yes	No	Yes	No	No	No	No	No	No	No	No
Muformat	Yes	No	No	No	No	No	No	No	No	No	No	Yes	No
Mugamma	Yes	No	No	—	Yes	—	—	No	Yes	No	No	Yes	No
muJava	Yes	Yes	No	No	Yes	No	No	No	Yes	Yes	No	No	No
Proteum	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Sesame	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
Testooj	Yes	Yes	No	—	Yes	Yes	—	No	No	No	No	No	No

* G&E = generation and execution.

suite. Use good software testing techniques, try to avoid randomness in your tests, make them reproducible whenever you can, and keep test cases independent.

- Perform mutation analyses at night. Executing tests against mutants can take a lot of time, but it's an unattended task.
- Select a proper threshold for the mutation score. Reaching a 100 percent mutation score has more

costs than benefits. A threshold of 85 or 90 percent is usually enough, but take into account that effectiveness might decrease.

Consider trying mutation the next time you need to release a software project. Use it in the most critical parts of your system. You might be surprised with the excellent quality of your test suites!

Acknowledgments

This work is partially supported by the GEODAS project (Spanish Ministry of Economy and Competitiveness and European Fund for Regional Development, TIN2012-37493-C03-01). The authors thank Christof Ebert for his very valuable suggestions.

References

1. M. Polo and P. Reales, "Mutation Testing Cost Reduction Techniques: A Survey," *IEEE Software*, vol. 27, no. 3, 2010, pp. 80–86.
2. R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, 1978, pp. 34–41.
3. R.H. Untch, A.J. Offutt, and M.J. Harrold, "Mutation Analysis Using Mutant Schemata," *Proc. 1993 ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, 1993, pp. 139–148.
4. Y.-S. Ma, J. Offutt, and Y.R. Kwon, "MuJava: An Automated Class Mutation System: Research Articles," *Software Testing Verification Reliability*, vol. 15, no. 2, 2003, pp. 97–133.
5. A.J. Offutt and S.D. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Trans. Software Eng.*, vol. 20, 1994, pp. 337–344.
6. M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Software Testing Verification Reliability*, vol. 19, no. 2, 2009, pp. 111–131.

PEDRO REALES is a researcher at the University of Castilla–La Mancha. His research interests include software testing and software test automation. Reales received a PhD in computer science from the University of Castilla–La Mancha. Contact him at pedro.reales@uclm.es.

MACARIO POLO is an associate professor of computer science at the University of Castilla–La Mancha. His research interests include software testing and software engineering automation. Polo received a PhD in computer science from the University of Castilla–La Mancha. Contact him at macario.polo@uclm.es.

JOSÉ LUIS FERNÁNDEZ-ALEMÁN is a professor of computer science at the University of Murcia. His research interests include mutation testing, computer-based learning and its application to computer science, and nursing. Fernández-Alemán received a PhD in computer science from the University of Murcia. Contact him at aleman@um.es.

AMBROSIO TOVAL is a full professor of computer science at the University of Murcia. His research interests include software engineering—specifically, requirements engineering. Toval received a PhD in computer science from the Universidad Politécncia de Valencia. Contact him at atoval@um.es.

MARIO PIATTINI is a full professor of computer science at the University of Castilla–La Mancha. His research interests include global software development and green software. Piattini received a PhD in computer science from the Universidad Politécnica de Madrid. Contact him at mario.piattini@uclm.es.