feature

testing

Mutation Testing Cost Reduction Techniques:

A Survey

Macario Polo Usaola and Pedro Reales Mateo, *University of Castilla-La Mancha*

Although mutation's main steps (mutant generation, test case execution, and result analysis) can be costly, research allows developers to apply it to industry.

From the research perspective, mutation is a mature testing technique that has often shown its value for evaluating both software and software testing techniques. However, to the best of our knowledge, there's an important gap between its current research status and the possibilities of adopting it for the industrial world, owing to its high costs.

For three decades, researchers have made considerable effort and obtained sufficient results regarding mutation. However, neither software practitioners nor testing-tool developers have put the results to work. Here, we describe research on cost reduction in mutation testing, focusing on techniques that could easily transfer to industrial practice.

Preliminary concepts

Richard DeMillo and his colleagues proposed mutation as a testing technique in 1978.[1] They describe this basic idea as follows:

> *A programmer enters from a terminal a program, P, and a proposed test data set whose adequacy is to be determined. The mutation system first executes the program on the test data: if the program gives incorrect answers then certainly the program is in error. On the other hand, if the program gives correct answers, then it may be that the program is still in error, but the test data is not sensitive enough to distinguish that error: it is not adequate. The mutation system then creates a number of mutations of P that differ from P only in the occurrence of simple errors.*[1]

So, a mutant *M* of a program under test *P* is a copy of *P* that contains a small code change that's interpreted as a fault. Mutation relies on the ability of the test data set (the *test suite*) to find faults in the set of mutants.

Test engineers typically use automated tools to generate mutants. These tools apply a set of mutation operators to *P*. They define each mutation operator to introduce some type of syntactic change to a statement. For example, a simple instruction such as `return a + b` (where *a* and *b* are integers) can mutate in at least 20 different ways ($a - b$, $a \times b$, a / b, $a + b{+}{+}$, $-a + b$, $a + -b$, $0 + b$, $a + 0$, $|a| + b$, $a + |b|$, and so on), depending on the mutation operators. Thus, the number of mutants generated even for a medium-size program can be very large. Dealing with this number of mutants has implications regarding the time needed to compile, link, and execute them.

Automated tools typically execute test cases against the original program and the mutants, registering the results with each program version (original or mutant). When the result of executing a test case against a mutant *M* differs from the same test case against *P*, the test case has found the fault introduced in *M*, and the mutant is *killed;*

0740-7459/10/$26.00 © 2010 IEEE

otherwise, the mutant is *alive*. Thus, the mutation testing aims "to kill all the mutants." An *oracle* compares states of the mutant and the original program after executing each test case. Here, the main problem is the number of executions required to execute all the test cases against all the mutants, which, in principle, should be $|T| \times |M|$, where T is the set of test cases and M is the set of mutants (you must also consider $|T|$ additional executions for the original program).

Many mutants that remain alive will never be killed because they're *equivalent mutants* and will always produce the same output as P for any test case. Actually, the "fault" introduced in equivalent mutants isn't a fault but an optimization or deoptimization of the code (for example, the Java instructions `return a` and `return a++` provide the same result). Equivalent mutants are really noise and make the third step of mutation testing difficult—analyzing test case execution results. Taking into account the set of equivalent mutants, Equation 1 gives the quality of a test suite (measured in terms of the number of mutants killed) and defines the *mutation score*:

$$MS(P,T) = \frac{K}{(M - E)}, \quad (1)$$

where P is the program under test; T is the test suite; K is the number of mutants killed; M is the number of mutants generated; and E is the number of equivalent mutants.

So, mutation testing's main difficulties come from the number of mutants the operators generate, the number of required executions, and the result analysis step, which is hampered by the equivalent mutants introduced, usually around 20 percent. These difficulties, together with the "strange" nature of mutation (discovery of artificially seeded faults), mean that this testing technique hasn't received much attention from the industrial community, which is more interested in detecting real faults in the actual application. Some studies have discussed how discovering all the faults seeded by mutation operators might subsume (see the "Subsumption of Coverage Criteria" sidebar),[2] *probably subsume*,[3] or correspond to several widely accepted coverage criteria (such as decision, condition, condition/decision, and modified decision/condition). From here, you can consider the mutation score as an adequate-coverage criterion if good mutation operators are applied.[4] Indeed, killing a set of mutants generated with a good set of operators helps to fulfill two goals. The first is to have a good test suite, T (if T discovers all the artificial faults). The second is to have a reliable original program P, if T finds no faults in P.

Subsumption of Coverage Criteria

A criterion coverage C_1 subsumes another criterion C_2 if for every program, any test set T that satisfies C_1 also satisfies C_2.[1] Three examples of such subsumption follow. Exercising all statements in a class subsumes all methods. If a test suite traverses all the edges in a connected graph, all its nodes are also traversed. So, all-edges subsumes all-nodes. The decision coverage criterion requires that each decision execute at least once. The logical connector replacement mutation operator replaces each decision in a program by true and false. To kill them, test cases respectively taking the false and the true branches must be written to cover decisions.[2]

References

1. P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.*, vol. 14, no. 10, 1998, pp. 1483–1498.
2. A.J. Offutt and J.M. Voas, *Subsumption of Condition Coverage Techniques by Mutation Testing*, tech. report ISSE-TR-96-01, Dept. of Information and Software Systems Eng., George Mason Univ., 1996.

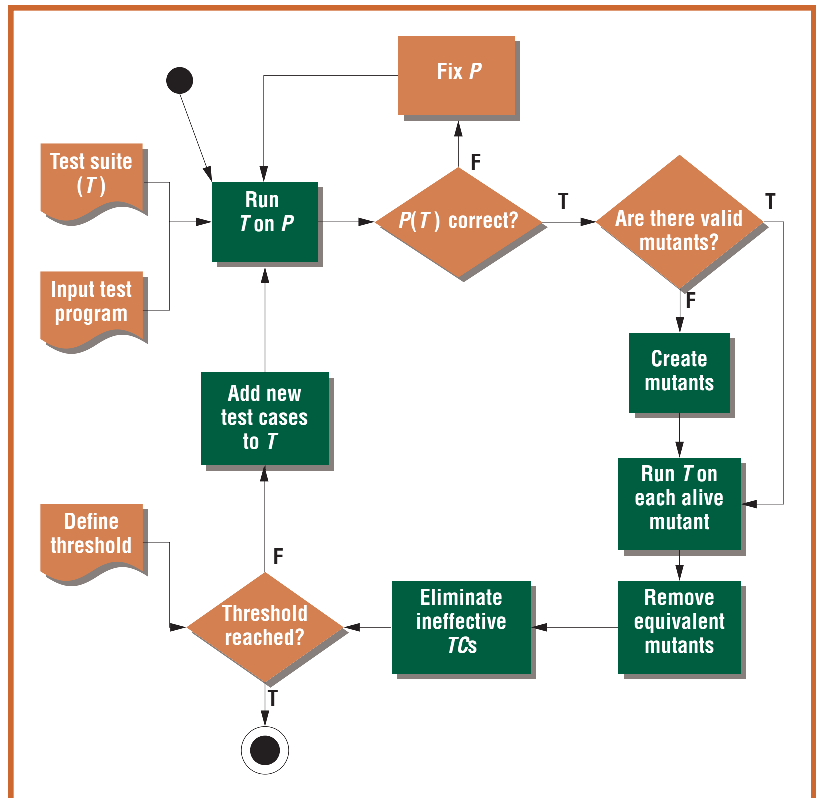


Figure 1. A modified version of mutation testing where *T* is the test suite, *P* is the program under test, and *TC* is a test case. The tester checks the correct behavior of the original program before generating mutants.

Mutation Testing

Figure 1 shows a testing process that slightly modifies the one that A. Jefferson Offutt proposed.[5]

100 percent mutation score for 10 percent of the mutants is nearly adequate for a full mutation analysis.

Offutt's process proposes generating mutants and iteratively executing the test cases against the living ones. As long as the process doesn't reach a minimal, preestablished mutation score threshold, the tester must add new test cases to the suite until he or she finds the desired number of introduced faults. Then, the tester compares the results of executing the test cases against the original program with the expected results. If any are incorrect, the developers must fix the original program and restart the process. So, it's possible to detect errors in the original program after having achieved the desired mutation score, which may require a new execution of all the steps. Once the original program has changed, some or all of the first-generation mutants are no longer valid, so the tester (perhaps using a tool) must create and execute new test cases. Then the tester, assisted by the mutation tool, should analyze the results again.

To mitigate this possibility, the tester should first execute the test cases against the input program to find any faults as soon as possible. In the refined process in Figure 1, the tester evaluates the correction of the input program for the initial test suite. It's obviously important to write a good set of test cases that provide as much coverage as possible. To this end, the literature includes a broad set of test data selection techniques and combination strategies to produce good test suites.[6–7] (For additional resources, tools, and so on, see Mutation Testing Online; www.mutationtest.net.)

Once the tester has a process model to follow, the next point of interest is reducing costs in the "create mutants," "run *T* on each alive mutant," and "threshold reached" boxes of the figure. Eliminating ineffective test cases can occur during test case execution or after, by applying a test-suite-reduction algorithm based on mutation.

Mutant Generation

When mutants are generated, several operators can mutate almost every executable instruction in the original program, meaning that the number of mutants generated for a normal program can be huge. Depending on the system, this can result in high costs for compilation and further steps. To reduce such costs, the tester can select either random mutants or the best mutation operators (*selective mutation*).

Regarding random selection, research has shown that a 100 percent mutation score for 10 percent of the mutants is nearly adequate for a full mutation analysis.[8–11] However, such selection requires generating, compiling, and linking all the mutants.

Researchers have concentrated on selective mutation, which consists of generating mutants using only a reduced subset of mutation operators. The criterion for selecting the operators lies in the "goodness" of the mutants generated. Elfurjani S. Mresa and Leonardo Bottaci conducted an empirical study to determine the best operators and when it's preferable to use random selection.[6] They observed, for example, that SVR (scalar variable replacement), ASR (array reference for scalar variable replacement), and CSR (constant for scalar variable replacement) operators generate the most mutants (confirming a previous analysis[12]) and that these operators perhaps shouldn't be included in a selective set. Mresa and Bottaci classify the mutation operators in several categories. They reached two main conclusions.

- If the program under test requires a mutation score very close to 100 percent, then random selection is more efficient than selective mutation.
- If less stringent test coverage is acceptable, then selective mutation based on a restricted set of efficient operators—AOR (arithmetic operator replacement), SAN (statement analysis), SDL (statement deletion), ROR (relational operator replacement), and UOI (unary operation insertion)—is more efficient.

In a previous study, Offutt and his colleagues concluded that test sets that are adequate for the mutants generated by AOR, ROR, UOI, ABS (absolute value insertion), and LCR (logical connector replacement) achieve a full mutation score of 99 percent, reducing the number of mutants generated by 77 percent.[13]

The suitability of mutation operators for testing a program might depend on its programming language. The selected operators apply to almost any programming language; however, they don't consider, for example, the manipulation of pointers in languages such as C or C++ or the characteristics of object orientation, such as inheritance and polymorphism. In this respect, James Andrews and his colleagues conducted an experiment on C programs by applying Offutt's selected operators and adding SDL because the subject programs "contained a large number of pointer-manipulation and field-assignment statements that would not be vulnerable to any of the sufficient mutation operators."[14]

Mutation operators try to imitate common errors that programmers commit (such as using a null pointer or not overriding an inherited operation) and rely on the *coupling effect*, in which

a test data set that detects all simple faults in a program is so sensitive that it also detects more complex faults.[15–16] One criticism of mutation testing is the artificial nature of the faults seeded. Andrews and his colleagues reached two important conclusions. First, using selectively generated mutants (from which the equivalent mutants must be removed) can indicate a test suite's fault detection ability. Second, their experiment "shows the danger of using faults selected by humans, since it leads to underestimating the fault detection ability of test suites." Thus, they also note the convenience of automatic mutant generation, which provides a well-defined, fault-seeding process and the possibilities of replication and criteria subsumption.[14]

In a study regarding sufficient mutation operators for C, Ellen F. Barbosa and her colleagues selected from the mutation operators in the Proteum tool this set of operators: SWDD (while replacement by do-while), SMTC (*n*-trip continue), SSDL (statement deletion), OLBN (logical operator by bitwise operator), OASN (arithmetic operator by shift operator), ORRN (relational operator mutation), VTWD (twiddle mutations), VDTR (domain traps), Cccr (constant for constant replacement) and Ccsr (constant for scalar replacement).[17]

In addition, Yu-Seung Ma and her colleagues have developed MuJava, a Java mutation testing tool that uses *mutant schemata generation* to directly manipulate Java bytecode, thus saving time in mutant compilation.[18]

Test Case Generation and Execution

Having a reduced-size test suite is important, especially for regression testing during software maintenance. Mats Grindal and his colleagues reviewed strategies for test case generation, each with its advantages and drawbacks: *Each choice,* for example, produces small test suites but provides low coverage. *All combinations* provides the highest coverage but produces the largest test suites. Moreover, many of those cases are redundant, because they don't increase the coverage reached by other test cases in the same suite.[7]

Regarding test case execution, the most common way to eliminate the ineffective test cases (see the box in Figure 1) is to execute each test case only against the mutants that remain alive. So, after the execution, the tester obtains a reduced test suite that reaches the same mutation score as the whole test suite. Consider, for example, Table 1, which shows a program with seven mutants and a test suite with six test cases. At first glance, the complete execution requires 6 × 7 = 42 executions (setting aside the six executions in the original program). If test cases execute only against those mutants that remain alive, then the number of executions might decrease significantly: *tc*1 kills *m*1 and *m*2, which are removed from the mutant suite (there are seven executions at this point). Then, *tc*2 executes against *m*3 to *m*7 (five executions), and *m*3 is removed from the mutant suite because it's killed. Then, *tc*3 executes on *m*4 to *m*7, removing *m*4, *m*5, and *m*6 from the mutant set. In this example, *tc*4 and *tc*5 execute with no positive results against the only live mutant (*m*7). Finally, one more execution of *tc*6 kills *m*7. In this way, only 19 test case executions are required instead of 42, and the test suite can be reduced to four test cases.

Table 1

A killing matrix for a supposed program

Mutant	Each X represents that the tc_i test case has killed the m_j mutant					
	*tc*1	*tc*2	*tc*3	*tc*4	*tc*5	*tc*6
*m*1	X	X				
*m*2	X	X			X	
*m*3		X				X
*m*4			X			X
*m*5			X			X
*m*6			X			X
*m*7						X

Another possibility is reducing the suite after all test cases execute. Although the problem of minimizing a test suite (the "optimal test-suite reduction problem") has been shown to be NP-hard[19] (and thus has no solution in polynomial time), several approaches present greedy algorithms for its solution (along with several authors, Neelam Gupta has worked intensively in this area[20]). These approaches require complete execution of all test cases against all the mutants: in Table 1, *tc*1 and *tc*6 reach the same mutation score as the complete test suite. If the test case selection occurs during mutant execution (as in the example given in the previous paragraph, where four test cases were selected), the reduced suite can be farther from the minimum size obtained by a greedy algorithm (as in this example, where only two test cases are selected). Since testing is often programmed as

Combining these techniques means a cost savings that could surpass 75 percent of the original costs.

unattended, nightly batch processes, the complete execution and further application of a greedy algorithm is a good choice for approaching the optimal reduced suites, which can be an important benefit in regression testing.

Another promising approach in test case execution is *weak mutation*, [21–23] which requires the continuous observation of the mutant being executed to check its intermediate state changes with respect to the original program. Classic mutation (also called *strong mutation*) considers a test case that kills a mutant when the test case output differs after executing it on the original and the mutant. More formally, this requires three conditions: reachability (the mutated statement must be reached), necessity (once the statement has been reached, the test case must cause an erroneous state on the mutant), and sufficiency (the erroneous state must be propagated to the output). Weak mutation only requires the two first conditions to detect the change introduced in the mutant, considering it's killed just when the different state is detected.

Result Analysis

The most important obstacle in this third step is the presence of equivalent mutants. Phyllis Frankl and her colleagues discuss the almost prohibitive cost of detecting equivalent mutants.[24] Bernhard Grün and his colleagues report of a duration of 15 minutes to assess the equivalence of a single mutation.[25]

From a formal point-of-view, the problem with detecting all equivalent mutants is undecidable, although in practice you can detect some by annotating the program under test with restrictions[26] and program slicing.[27] However, the industry doesn't usually apply these techniques, so they aren't easily adaptable to common software development practice.

Many selective-mutation concepts aim to reduce the number of equivalent mutants, which implies a considerable reduction during result analysis. From the automatable-techniques perspective, a recent paper discusses perhaps the most significant results and relies on n-order mutation.[28] An n-order mutant contains n faults instead of 1 and proceeds from a previous generation's combination of mutants. Thus, two first-order mutants (each with a fault) are combined into a second-order mutant with two faults, which might in turn be combined with another first-order mutant to obtain a third-order mutant.

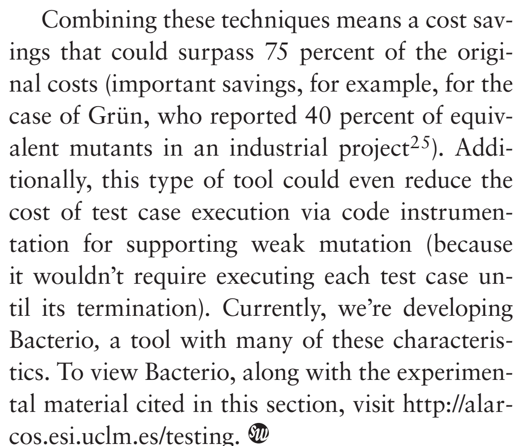
The paper describes three algorithms for producing second-order mutants from first-order mutants and shows meaningful cost reductions in mutation testing, especially during result analysis. At first glance, the number of second-order mutants corresponding to a set of first-order mutants is one-half (although seach algorithm produces different quantities of second-order mutants). In general, there will be $1/n$ n-order mutants. The number of test case executions also decreases (against $1/n$ instead of against n mutants). Perhaps more important, the percentage of equivalent mutants significantly decreases because with about 20 percent of first-order equivalent mutants, the probability of combining two equivalent mutants to produce a new one decreases to 4 percent. Obviously, the counterpart is the possibility of killing all the n-order mutants with test cases that only discover one of the n seeded faults. The authors of the paper include an experimental study with benchmark programs and some pieces of industrial software, concluding that, as long as the tester is aware of this risk, even sixth-order mutation can be effective.[28]

An industrially applicable mutation-testing tool should have these requirements:

- Users should be able to generate mutants with a selective set of generally applicable mutation operators, most likely AOR, ROR, UOI, ABS, and LCR. Additionally, and for specific languages or environments, the tool should consider including other concrete operators.
- Users should be able to select a random set of mutants.
- Also depending on the specific environment, the tool should allow mutation at compiled-code level (bytecode for Java, Microsoft Intermediate Language for .NET, and so on).
- In test execution, the tool should support both executing test cases on only the mutants remaining alive and, regarding batch, unattended testing cycles, selecting a reduced test suite with, for example, a greedy algorithm.
- The tool should support instrumentation of both the original program and the mutants to keep a log of the execution. Changes in a log would highlight a behavior difference, meaning that the corresponding mutant has been killed, and making this technique a type of weak mutation.
- To reduce result analysis costs, the tool should allow n-order mutation, which is easily automatable and transferable to industry.

Data from some experiments shows that a mean of 18.66 percent of the mutants generated are equivalent.[28] Taking the triangle-type problem as a possible baseline (a small program which many researchers have used for testing experiments), the MuJava tool generated 309 mutants (70 of them equivalent, 22.65 percent):

- Applying selective mutation could reduce the number of mutants by three-fourths—78 mutants. As we've discussed, there's significant confidence that a test suite killing these 78 mutants would also kill the original 309.
- Supposing a uniform distribution of equivalent mutants per operator (which actually isn't true, because each operator has a different proneness to produce this kind of noise), the process would generate 16 equivalent mutants.
- Combining the 78 selected mutants with a good combination algorithm (DifferentOperators is the best of the three presented) would produce between 50 and 55 percent second-order mutants (39 to 43), with about 5 percent (2) of equivalent mutants.



Combining these techniques means a cost savings that could surpass 75 percent of the original costs (important savings, for example, for the case of Grün, who reported 40 percent of equivalent mutants in an industrial project[25]). Additionally, this type of tool could even reduce the cost of test case execution via code instrumentation for supporting weak mutation (because it wouldn't require executing each test case until its termination). Currently, we're developing Bacterio, a tool with many of these characteristics. To view Bacterio, along with the experimental material cited in this section, visit http://alarcos.esi.uclm.es/testing.

Acknowledgments

The PRALÍN (Pruebas en Líneas de Producto, Junta de Comunidades de Castilla-La Mancha/ European Social Fund, grant PAC08-121-1374) and the PEGASO/MAGO (Ministerio de Ciencia Innovación, grant TIN2009-13718-C02-01) projects partially supported this work.

References

1. R. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, 1978, pp. 34–41.
2. A.J. Offutt and J.M. Voas, *Subsumption of Condition Coverage Techniques by Mutation Testing*, tech. report ISSE-TR-96-01, Dept. of Information and Software Systems Eng., George Mason Univ., 1996.
3. A.J. Offutt et al., "An Experimental Evaluation of Data Flow and Mutation Testing," *Software: Practice and Experience*, vol. 26, no. 2, 1996, pp. 165–176.
4. M. Polo, M. Piattini, and S. Tendero, "Integrating Techniques and Tools for Testing Automation," *Software Testing, Verification and Reliability*, vol. 17, no. 1, 2007, pp. 3–39.
5. A.J. Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer," *Proc. 12th Int'l Conf. Testing Computer Software* (ICST 95), IEEE CS Press, 1995, pp. 99–109.
6. E.S. Mresa and L. Bottaci, "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, 1999, pp. 205–232.
7. M. Grindal, A.J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, 2005, pp. 167–199.
8. R.A. DeMillo and E.H. Spafford, "The Mothra Software Testing Environment," *Proc. 11th NASA Software Eng. Laboratory Workshop*, Goddard Space Center, 1986.
9. A.T. Acree, "On Mutation," doctoral dissertation, School of Information and Computer Science, Georgia Inst. of Technology, 1980.
10. R.A. DeMillo et al., "An Extended Overview of the Mothra Software Testing Environment," *Proc. 2nd Workshop Software Testing, Verification, and Analysis*, IEEE CS Press, 1988, pp. 142–151.
11. K.N. King and A.J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software: Practice and Experience*, vol. 21, no. 7, 1991, pp. 685–718.
12. A.P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *Proc. 15th Ann. Int'l Computer Software and Applications Conf.*, IEEE CS Press, 1991, pp. 604–605.
13. A.J. Offutt et al., "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, 1996, pp. 99–118.
14. J. Andrews, L. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. 2005*

About the Authors

Macario Polo Usaola is a professor of computer science in the Department of Information Systems and Technologies at the University of Castilla-La Mancha. He's also an active member of the Alarcos Research Group. His research interests relate to the automation of software testing tasks. Polo has a PhD in computer science from the University of Castilla-La Mancha. Contact him at macario.polo@uclm.es.

Pedro Reales Mateo is a PhD student of computer science in the University of Castilla-La Mancha's Department of Information Systems and Technologies. His research interests relate to the automation of software testing. Reales has an MSc in computer science from the University of Castilla-La Mancha. Contact him at pedro.reales@uclm.es.

Int'l Conf. Software Eng. (ICSE 05), ACM Press, 2005, pp. 402–411.
15. R. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, 1978, pp. 34–41.
16. A.J. Offut, "Investigations of the Software Testing Coupling Effect," *ACM Trans. Software Eng. and Methodology*, vol. 1, no. 1, 1992, pp. 15–20.
17. E.F. Barbosa et al., "Toward the Determination of Sufficient Mutant Operators for C," *Software Testing, Verification and Reliability*, vol. 11, no. 2, 2001, pp. 113–136.
18. Y.-S. Ma, "MuJava: An Automated Class Mutation System," *Software Testing, Verification and Reliability*, vol. 15, no. 2, 2005, pp. 97–133.
19. M.R. Garey and D.S. Johnson, *Computers and Intractability*, W.H. Freeman, 1979.
20. D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," *Proc. 21st Int'l Conf. Software Maintenance* (ICSM 05), IEEE CS Press, 2005, pp. 549–558.
21. R. DeMillo, E. Krauser, and A. Mathur, "Compiler-Integrated Program Mutation," *Proc. 15th Ann. Computer Software and Applications Conf.* (Compsac 91), pp. 351–356. 1991.
22. W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.*, vol. 8, no. 4, 1982, pp. 371–379.
23. A.J. Offutt and S.D. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Trans. Software Eng.*, vol. 20, no. 5, 1994, pp. 337–344.
24. P.G. Frankl, S.N. Weiss, and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *J. Systems and Software*, vol. 38, no. 3, 2007, pp. 235–253.
25. B.J.M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation Workshops* (ICST 09), IEEE CS Press, 2009, pp. 192–199.
26. A.J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, 1997, pp. 165–192.
27. R. Hierons and M. Harman, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, 1999, pp. 233–262.
28. M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, 2008, pp. 111–131.

cn Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

Running in Circles Looking for a Great Computer Job or Hire?

The IEEE Computer Society Career Center is the best niche employment source for computer science and engineering jobs, with hundreds of jobs viewed by thousands of the finest scientists each month - **in *Computer* magazine and/or online!**

- Software Engineer
- Member of Technical Staff
- Computer Scientist
- Dean/Professor/Instructor
- Postdoctoral Researcher
- Design Engineer
- Consultant

http://careers.computer.org

The IEEE Computer Society Career Center is part of the *Physics Today* Career Network, a niche job board network for the physical sciences and engineering disciplines. Jobs and resumes are shared with four partner job boards - *Physics Today* Jobs and the American Association of Physics Teachers (AAPT), American Physical Society (APS), and AVS: Science and Technology of Materials, Interfaces, and Processing Career Centers.