In Search of What We Experimentally Know about Unit Testing

Natalia Juristo, Ana M. Moreno, and Sira Vegas, *Technical University of Madrid*

Martín Solari, *ORT University Uruguay*

Experimental studies that compare testing techniques, test-set quality, and the kinds of faults detected are aggregated to identify empirically grounded information about unit testing.

Gathering evidence in any discipline is a lengthy procedure, requiring experimentation and empirical confirmation to transform information from mere opinion to undisputed fact. Software engineering is a relatively young field and experimental SE is even younger, so undisputed facts are few and far between. Nevertheless, ESE's relevance is growing because experimental results can help practitioners make better decisions.[1,2]

Aggregating experimental results derived from individual empirical studies that address the same problem is an important task—and a grueling one. It involves tough challenges such as identifying relevant experiments, understanding and assessing them to choose relevant results, and combining and synthesizing the results.[3] Aggregation is not easy because separate experiments have different settings and subjects, tend to assess different variables, and are usually trying to optimize different phenomena.[4,5] That's why aggregation results are not always as conclusive as we may desire.

We've aggregated results from unit-testing experiments with the aim of identifying information with some experimental basis that might help practitioners make decisions. We made an extensive search in IEEE Xplore and ACM digital libraries looking for publications of such experiments. Most of the experiments focus on two important characteristics of testing techniques: *effectiveness* (number of faults found) and *efficiency* (effort required to apply the technique). Some other experiments study the quality of test-case sets according to different criteria. The aggregation results are far from ideal. Most of the studies represent laboratory experiments conducted on students using nonindustrial software, and some would benefit from further statistical analysis. The studies should be replicated in industrial settings to extend the generalizations derived from them. Even so, we've synthesized some evidence that we believe can be useful to practitioners. The results advance our discipline's maturity beyond mere opinions.

Although we've tried to include as many experiments as possible, it's practically impossible to ensure completeness, and we might have missed some. Because we're aggregating experimental results, we didn't consider either theoretical studies or unit-testing simulations—that is, studies that focused on a technique's underlying theory to derive its per-

0740-7459/06/$20.00 © 2006 IEEE

formance or reliability-related characteristics. The experiments cover three testing areas: test-case generation, test-set evaluation, and test-case selection. There are many approaches to each of these tasks and several ways to classify them. We've developed one possible breakdown, which appears in the "Unit-Testing Techniques" sidebar and tables (pp. 73–76). The tables cross-reference the techniques to the related experiments. For reasons of space, a detailed description of each experiment is impracticable here but available elsewhere.[6] We've organized our findings according to the three testing areas the experiments cover.

Findings on test-case generation

Experiments that address test-case generation mainly compare the efficiency and effectiveness (and sometimes the fault type) of specification-based, code-based, and fault-based techniques.

Let's start with the experiments that compare the effectiveness of specification-based techniques with code-based techniques that use control-flow criteria. *Specification-based techniques* generate test cases on the basis of program inputs, input/output relations, or the possible program states.[7] *Control-flow-based criteria* vary as to how well the test cases cover the program's control flow (see the sidebar, table A).

The experimental results indicate that specification-based techniques are generally more effective than code-based techniques that use criteria with weak coverage levels. More specifically, experimenters found that boundary-value analysis is more effective than statement coverage but takes longer and needs experienced testers.[8] However, when time constraints exist and testers are inexperienced, statement coverage can be more effective. Another study also concluded that boundary-value analysis takes less time than condition coverage, whereas its effectiveness is similar.[9]

The testing literature has assumed that specification-based and code-based techniques complement each other in the kind of faults they detect.[10] Some experiments support this supposition, showing that boundary-value analysis detects different faults from sentence coverage and branch coverage.[9,11]

Experimenters have also studied whether testers applying different techniques (specification based and code based one after the other) find more faults than testers applying a single technique. In particular, one experiment considered boundary-value analysis and branch coverage,[11] finding a higher number of faults combining techniques than using only one.

Some experimental results have confirmed intuitive practitioner knowledge. For example, the fault-detection effectiveness of boundary-value analysis, sentence coverage, predicate coverage, and branch coverage depends on the program[8,9,11] and varies according to the tester's experience in creating test cases.[8]

All these results for control-flow criteria come from experiments conducted on small programs, and all but one[8] used student testers; so, we can generalize about them only with caution.

Data-flow-based criteria are another way to generate code-based test cases (see the sidebar, table A). Such test cases cover the execution space between where variables are defined and where they're used. One study analyzed the number of test cases generated with the *all-du-paths* data-flow criterion on nonindustrial programs.[12] (The all-du-paths criterion requires the test set to traverse every simple definition-use path between a variable and its use in the program.) This study suggested that the all-du-paths criterion is practical because it doesn't generate as many test cases as literature had suggested.[13]

Another experiment compared the effectiveness of data-flow versus control-flow criteria.[14] Neither criteria proved irrefutably more effective in generating test cases than the other. Instead, performance appears to depend on the program on which they are applied. However, as you might expect, some indicators suggest that, at least in the small programs ex-

Unit-Testing Techniques

Testing techniques follow different strategies or criteria to create, evaluate, or select a set of inputs to run a program that looks for faults. Finding a standard way of pigeonholing all of them isn't easy. So, we've made tables A (pp. 74–75) and B (p. 76) based on the Software Engineering Body of Knowledge (SWEBOK) classification (www.swebok.org). For a detailed study of these techniques, we refer readers to the classical testing literature.

Some techniques in table A can be used for both generating and evaluating test sets; when a technique is not applicable, n/a appears in the corresponding cell. The other cells include cross-references to the experiments discussed in the main article or an em-dash (—) if we found no experiments relative to the technique.

Table A

Techniques for test-set generation and test-set evaluation, conforming to the SWEBOK classification scheme

Technique type	Technique		Description	Test-set generation experiments	Test-set evaluation experiments
Based on tester's intuition and experience	Ad hoc testing		The program is exercised without any specific guidelines.	—	n/a
	Exploratory testing		Test cases are designed and dynamically modified on the basis of the tester's experience.	—	n/a
Specification based	Equivalence partitioning		A test case is generated for each equivalence class found. The test case is selected at random within the class.	—	n/a
	Boundary-value analysis		Several test cases are generated for each equivalence class: one that belongs to the inside of the class and as many as necessary to cover its limits (boundaries).	V.R. Basili and R.W. Selby,[8] E. Kamsties and C.M. Lott,[9] M. Roper et al.[11]	n/a
	Decision tables/ cause-effect graphing		Test cases are derived considering every possible combination of inputs and outputs.	—	n/a
	Finite-state-machine-based		Test cases are selected to cover states and transitions of a state machine that models the program.	—	n/a
	Derived from formal specifications		Test cases are generated from formal specification languages.	—	n/a
	Random testing		Test cases are generated at random according to the input domain defined in the specification.	M. Hutchins et al.[14]	n/a
Code based	Control-flow-based criteria	Sentence coverage	Test cases must ensure that all program sentences are executed at least once.	V.R. Basili and R.W. Selby[8]	—
		Decision (branch) coverage (all edges)	Test cases must ensure that all program decisions take the value true and false.	M. Roper et al.,[11] M. Hutchins et al.[14]	P. Frankl and O. Iakounenko,[20] P.G. Frankl and Y. Deng,[21] P.G. Frankl and S.N. Weiss[22]
		Condition (predicate) coverage	Test cases must ensure that all conditions (predicates) forming the logical expression of the decisions take the value true and false.	E. Kamsties and C.M. Lott[9]	—
		Decision/ condition coverage	Test cases with condition coverage are supplemented to achieve decision coverage.	—	—
		Modified condition/ decision coverage	Test cases must ensure that the independent influence of every condition on the decision is tested (every condition is varied while holding fixed all other possible conditions).	—	—
		Reinforced condition/ decision coverage	The modified condition/decision rule is required, and every condition must independently keep the decision value to true and false.	—	—
		Path coverage	Test cases must execute all program paths. This criterion isn't workable in practice.	—	—
	Data-flow-based criteria	All-definitions	Test cases must cover each definition of each variable for at least one use of the variable.	—	—

Table A (continued)

Technique type	Technique		Description	Test-set generation experiments	Test-set evaluation experiments
Code based (cont'd)	Data-flow-based criteria (cont'd)	All-c-uses, some-p-uses	Test cases must ensure that there is at least one path of each variable definition to each use of the variable in a computation. If some variable definitions aren't covered, use p-uses.	—	—
		All-p-uses, some-c-uses	Test cases must ensure that there is at least one path of each variable definition to each use of the variable in a predicate. If some variable definitions aren't covered, use c-uses.	—	—
		All-c-uses	Test cases must ensure that there is at least one path of each variable definition to each use of the variable in a computation.	—	E.J. Weyuker[19]
		All-p-uses	Test cases must ensure that there is at least one path of each variable definition to each use of the variable in a predicate.	—	E.J. Weyuker[19]
		All-uses	Test cases are generated so that there is at least one path of each variable definition to each use of the definition.	E. Wong and A.P. Mathur[17]	E.J. Weyuker,[19] P. Frankl and O. Iakounenko,[20] P.G. Frankl and Y. Deng,[21] P.G. Frankl and S.N. Weiss,[22] P.G. Frankl et al.[23]
		All-du-paths	Test cases must execute all possible paths of each definition of each variable to each use of the definition.	J.M. Bieman and J.L. Schultz[12]	E.J. Weyuker[19]
		All-dus	Test cases must execute all possible executable paths of each definition of each variable to each use of the definition.	M. Hutchins et al.[14]	—
	Reference models for code-based testing (flow graphs, call graphs)		Test case generation is derived from the graphical representation of the programs.	—	n/a
Fault based	Error guessing		Test cases are generated guided by the subject's knowledge of what are typical faults.	—	n/a
	Mutation testing	Strong (standard) mutation	Test cases must cover all mutants generated by applying all mutation operators.	A.J. Offut and S.D. Lee,[15] A.J. Offut et al.,[16] E. Wong and A.P. Mathur,[17] H. Lee et al.[18]	P.G. Frankl et al.[23]
		Selective (constrained) mutation	Test cases must cover all mutants generated by applying some mutation operators.	A.J. Offut et al.,[16] E. Wong and A.P. Mathur[17]	—
		Weak mutation	Test cases must cover a given percentage of mutants generated by applying all mutation operators.	A.J. Offut and S.D. Lee,[15] E. Wong and A.P. Mathur[17]	—
		Schema-based mutation	All mutants for a program are encoded into one metaprogram to improve test-execution performance.	—	—
Usage based	Operational profile		Test cases are generated according to their probability of occurring in actual operation.	—	n/a
	Software reliability-engineered testing		Tests are designed according to reliability objectives, expected use, and criticality of different functions.	—	n/a
Fault seeding			Faults are artificially introduced in the program to evaluate test-set quality.	n/a	—

Table B

Test-set selection techniques, conforming to the SWEBOK classification scheme

Group	Technique		Description	Experiments
Filtering	Minimization		Redundant test cases are removed from the test set.	J.M. Kim et al.,[27] D. Leon et al.,[28] T.L. Graves et al.,[29] W.E. Wong et al.[33]
	Distribution based	Cluster filtering	Generated test cases are filtered on the basis of cluster analysis of execution profiles.	D. Leon et al.[28]
		Failure pursuit	Generated test cases are filtered on the basis of failure pursuit of execution profiles	D. Leon et al.[28]
Prioritization	No ordering		Generated test cases aren't ordered.	G. Rothermel et al.,[30] S. Elbaum et al.[31]
	Optimum		Generated test cases are ordered to optimize fault detection.	G. Rothermel et al.,[30,32] S. Elbaum et al.[31]; W.E. Wong et al.[33]
	Coverage based	Branch	Generated test cases are ordered according to their achieved branch coverage.	G. Rothermel et al.,[30] W.E. Wong et al.[33]
		Statement	Generated test cases are ordered according to their achieved statement coverage.	G. Rothermel et al.,[30] S. Elbaum et al.[31]
		Function	Generated test cases are ordered according to their achieved function coverage.	D. Leon et al.,[28] S. Elbaum et al.[31]
		FEP (fault-exposing potential)	Generated test cases are ordered according to the probability a modified statement will cause a failure for a given test case.	G. Rothermel et al.,[30] S. Elbaum et al.[31]
Regression	Random		Test cases are randomly selected from the existing test suite.	J.M. Kim et al.,[27] T.L. Graves et al.[29]
	Retest-all (traditional)		All tests in the existing test suite are run.	G. Rothermel and M.J. Harrold,[24] J. Bible et al.,[25] F. Vokolos and P.G. Frankl,[26] J.M. Kim et al.,[27] D. Leon et. al.,[28] T.L. Graves et al.[29]
	Safe	DejaVu[7]	Test cases are selected by detecting modified code entities such as functions, variables, types, and preprocessor macros.	G. Rothermel and M.J. Harrold,[24] J. Bible et al.,[25] J.M. Kim et al.,[27] T.L. Graves et al.[29]
		TestTube	Test cases are selected according to a variant of DejaVu that's less precise but more robust in analyzing large software systems.	J. Bible et al.,[25] J.M. Kim et al.[27]
		Textual differencing	Test cases are selected by comparing source files from the old and the new version of the program.	F. Vokolos and P.G. Frankl[26]
	Based on modifications		Test cases are selected to cover modified program components and those that the modifications might affect.	W.E. Wong et al.[33]
	Based on data-flow coverage		Test cases are selected to exercise data interactions that the modifications have affected.	T.L. Graves et al.[29]

amined, the effectiveness is greater if both criteria are applied together.[14]

Random testing is always the least costly strategy for generating test cases, so researchers have studied its effectiveness compared with testing based on data-flow and control-flow criteria. Laboratory results showed that all-edges and all-du-paths data-flow testing tended to be more effective than random testing; specifically, random testing must generate from 50 to 160 percent bigger test sets than all-edges and all-du-paths to be equally effective.[14] Because random generation is less costly, this size increase can sometimes be cost effective. However, you must consider execution costs as well as creation costs before opting for random testing over all-edges or all-du-paths.

Mutation testing is a *fault-based strategy* for generating test cases. Mutation operators represent the faults a program will likely contain. Testers use them to generate program variants containing source code errors, called *mutants*, which are used to generate a set of test cases. The sidebar table A shows possible mutation-testing variants.

Some experiments compared the effectiveness and number of test cases between standard mutation and its variants. In nonindustrial programs, some of these studies concluded that standard mutation is the most effective but also the most costly. Variants such as weak mutation and selective mutation are somewhat less effective; however, because they generate fewer mutants and so need fewer test cases, the variants also cost less to create. One study of weak mutation predicted that testers could apply it in place of standard mutation in noncritical systems without a sizeable drop in effectiveness.[15]Another study pointed out that the selective-mutation techniques, such as exweak/l or bb-weak/n, which generate fewer mutants and take less time, aren't much different in their effectiveness from standard mutation.[16] Other types of selective mutation, such as 10-percent-selective and abs/ror mutations, are faster but less effective.[17] A recent study applying standard mutation in the context of object-oriented programming found its effectiveness to be comparable to that of programs written in non-object-oriented languages.[18] The results showed few mutation operators applied to the classes and few mutants generated. However, all three of these selective-mutation studies would benefit from more statistical analysis.

An experiment run with nonindustrial software compared the effectiveness of mutation against data-flow criteria.[17] It's not yet clear whether a relevant difference exists between the techniques. In fact, the experiment suggested that abs/ror and 10-percent-selective mutations are about as effective as all-uses data-flow criteria.

Findings on test-set evaluation

Testers can also use testing criteria to evaluate test sets. As with test generation, the commonly used criteria are based on data and control flow, mutation, and their variants (see the sidebar, table A). Test-set evaluation measures the coverage level—that is, the *adequacy*—of the test sets for the chosen criteria. Many adequate test sets might exist for any one criterion, each with a different probability of finding faults. In experiments that compare criteria, researchers analyze the effectiveness and size of the adequate sets against such criteria.

One experiment studied the size of adequate data-flow test sets on small programs.[19] The results refuted the beliefs that all-uses criteria are preferable to all-p-uses and that all-du-paths criteria are preferable to all-uses. Instead, for test sets of similar size, the test sets that were adequate for the second criteria were also adequate for the first criteria. However, the researchers concluded that adequate all-c-uses test sets aren't also adequate for all-p-uses. This study also suggested that all-du-paths adequate test sets are linearly sized with regard to the number of program decisions. This result refutes the belief that the test sets would become exponentially larger.[13]

Other experiments examined the relationship between the coverage level and effectiveness of data-flow versus control-flow criteria. Using industrial software, some researchers found that the greater the coverage, the more effective the test set.[20,21] However, 100 percent coverage doesn't ensure 100 percent effectiveness, which other researchers have confirmed experimentally.[14] On the other hand, no one has irrefutably demonstrated that test sets with coverage based on data-flow techniques are more effective than those based on control flow, or vice versa. In fact, 50 percent of adequate all-uses test sets seem to be smaller than adequate all-edges test sets that have a similar effectiveness.[22] A later study in an industrial setting appears to support this result.[21] A deeper statistical data analysis of the data supporting this result could be useful to testing practitioners.

Researchers commonly use the null criterion (that is, no adequacy criterion applied to the test set) as a control group in comparing data-flow-adequate and control-flow-adequate test sets. For an adequacy study of test sets to be worthwhile, the effectiveness of the adequate test sets should be higher than the effectiveness of test sets using the null criterion. One study has suggested that the gain in effectiveness for adequate all-uses and all-edges sets is small with respect to the null criterion.[22] Again, more statistical analysis of this result would be beneficial. Nonetheless, we could infer that it's sometimes more cost effective simply to run all the test cases (the null criterion) than to conduct adequacy studies.

Another experiment run with nonindustrial software compared the effectiveness of mutation and all-uses-adequate test sets.[23] However,

Test-set evaluation measures the coverage level—that is, the adequacy—of the test sets for the chosen criteria.

the results didn't clearly establish differences in test-set effectiveness. More specifically, the researchers found standard mutation's effectiveness to be similar to that of all-uses, but it costs more to create the test sets with high coverage levels (which are always preferable) for mutation.

As the sidebar table A shows, some criteria apply to both generating and evaluating test sets. In these cases, testers can generate the test sets in two ways:

- *directly*, using the criterion for generation, or
- *randomly*, using the criterion for evaluation and further completion of the set until it achieves 100 percent coverage.

One study compared the size of test sets generated in these two ways for criteria based on data and control flows.[21] Direct generation produced smaller test sets. So, it might be worth analyzing the trade-off between the time used for creating, evaluating, and running the test cases.

Regression-testing techniques are the most widely used for selecting test cases from an existing set.

Findings on test-case selection

There are several approaches for selecting test cases from an existing set (see the sidebar, table B). Regression techniques temporarily select test cases from the set. Filtering is another way to reduce the number of test cases; however, it permanently removes the cases from the set. Finally, testing practitioners can prioritize test cases according to some criteria and can schedule them to execute the highest-priority cases first.

Regression-testing techniques are the most widely used for selection. Some experiments have studied the percentage of selected test cases and the time to select and run them. Laboratory experimentation has confirmed that, for small test sets, it's preferable to run the whole set without applying a regression technique. If the test-case set is large or if the program is extensive and takes a long time to run the set, the experimental results recommend applying safe reduction techniques, like DejaVu or TestTube. DejaVu appears to be preferable to TestTube for minimizing the number of cases.[24,25] Other experiments have analyzed the textual-differencing regression technique, but it's still not clear whether running all the cases is better than DejaVu.[26] For medium-sized sets, there are no conclusive results about which regression technique is best.

A few experiments have examined the effectiveness of *filtering techniques*. The results suggest that filtering might become more effective as the number of faults in a program increases,[27] although the data could benefit from further statistical analysis. Some research using industrial software compares minimization with distribution-based techniques, but no conclusive results have yet emerged.[28]

Researchers have also studied filtering together with regression techniques. More specifically, one experiment with nonindustrial programs concluded that modifications-based regression applied with minimization selects fewer cases without a big drop in effectiveness.[29]

Finally, *prioritization* studies include a few experiments that analyze fault-detection speed with industrial programs. Results show that fault-exposing potential (FEP) techniques are faster than statement-coverage techniques[30,31] and that statement-coverage techniques are faster (although more costly) than function coverage.[31] Another experiment studied test-case granularity in industrial programs, finding that granularity is likely to provide greater opportunities for prioritization, although this aspect of test-case selection needs further investigation.[32]

One experiment studied the percentage and precision of selected test cases by comparing the use of modifications-based regression along with minimization or prioritization.[33] For a nonindustrial setting, the researchers concluded that regression with minimization selects a lower percentage of test cases than regression alone or with prioritization. Furthermore, they found little precision loss for regression with prioritization or minimization.

The problems of aggregating testing and software engineering experiments are common to many branches of science, especially in disciplines dealing with highly variable settings involving people. The difficulties of controlling and quantifying the variables lead to disparities that rule out the application of formal meta-analysis techniques for aggregation, making comparisons and inferences harder.

In view of these complexities, as well as the shortage of SE experiments, our aggregation results aren't always as conclusive as we might desire. However, it's a step in the right direction. In the absence of empirical results, subjective opinions drive decision making. So, evidence gathered from experiments, even if it falls short of undisputed fact, can still help practitioners make better, grounded decisions.

Aggregating empirical results is also a way of confirming our intuitions. For example, our results include evidence that corroborates practitioners' instinctive beliefs about the dependency between testing techniques and programs and about different techniques for detecting different faults. Some of the evidence refutes common beliefs—for example, about the practical applicability of certain coverage criteria. Finally, the aggregation opens up new avenues for applied investigation, such as the different effects of several testers using the same technique or one tester applying several techniques.

References

1. T. Dyba, B.A. Kitchenham, and M. Jorgensen, "Evidence-Based Software Engineering for Practitioners," *IEEE Software*, Jan./Feb. 2005, pp. 58–65.
2. M.V. Zelkowitz, D.R Wallace, and D.W. Binkley, "Experimental Validation of New Software Technology," *Lecture Notes on Empirical Software Eng.*, N. Juristo and A.M. Moreno, eds., World Scientific, 2003, pp. 229–263.
3. N. Juristo and A.M. Moreno, *Basics of Software Engineering Experimentation*, Kluwer, 2001.
4. L.M. Pickard, B.A. Kitchenham, and P.W. Jones, "Combining Empirical Results in Software Engineering," *Information and Software Technology*, vol. 40, no. 14, 1998, pp. 811–821.
5. S.L. Pfleeger, "Soup or Art? The Role of Evidential Force in Empirical Software Engineering," *IEEE Software*, Jan./Feb. 2005, pp. 66–73.
6. N. Juristo, A.M. Moreno, and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments," *Empirical Software Eng.*, vol 9, no. 1, 2004, pp. 7–44.
7. *IEEE/ACM Software Eng. Body of Knowledge*, Iron Man Version, 2004; www.swebok.org.
8. V.R. Basili and R.W. Selby. "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.*, vol. 13, no. 2, 1987, pp. 1278–1296.
9. E. Kamsties and C.M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques," *Proc. 5th European Software Eng. Conf.* (ESEC 95), LNCS 989, Springer, 1995, pp. 362–383.
10. B. Beizer, *Software Testing Techniques*, Int'l Thomson Computer Press, 1990.
11. M. Roper, M. Wood, and J. Miller, "An Empirical Evaluation of Defect Detection Techniques," *Information and Software Tech.*, vol. 39, no. 11, 1997, pp. 763–775.
12. J.M. Bieman and J.L. Schultz, "An Empirical Evaluation (and Specification) of the All-du-paths Testing Criterion," *Software Eng. J.*, Jan. 1992, pp. 43–51.
13. E.J. Weyuker, "The Complexity of Data Flow Criteria for Test Data Selection," *Information Processing Letters*, vol. 19, no. 2, 1984, pp. 103–109.
14. M. Hutchins et al., "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), IEEE CS Press, 1994, pp. 191–200.
15. A.J. Offut and S.D. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Trans. Software Eng.*, vol. 20, no. 5, 1994, pp. 337–344.
16. A.J. Offut et al., "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, 1996, pp. 99–118.
17. E. Wong and A.P. Mathur, "Fault Detection Effectiveness of Mutation and Data-flow Testing," *Software Quality J.*, vol. 4, no. 1, 1995, pp. 69–83.
18. H. Lee, Y. Ma, and Y. Kwon, "Empirical Evaluation of Orthogonality of Class Mutation Operators," *Proc. 11th Asia-Pacific Software Eng. Conf.* (APSEC 04), IEEE CS Press, 2004, pp. 512–518.
19. E.J. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Trans. Software Eng.*, vol. 16, no. 2, 1990, pp. 121–128.
20. P. Frankl and O. Iakounenko, "Further Empirical Studies of Test Effectiveness," *Proc. Int'l Symp. Foundations Software Eng.*, ACM Press, 1998, pp. 53–162.
21. P.G. Frankl and Y. Deng, "Comparison of Delivered Reliability of Branch, Data Flow and Operational Testing: A Case Study," *Proc. ACM Int'l Symp. Software Testing and Analysis*, ACM Press, 2000, pp. 124–131.
22. P.G. Frankl and S.N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Trans. Software Eng.*, vol. 19, no. 8, 1993, pp. 774–787.
23. 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, 1997, pp. 235–253.

About the Authors

Natalia Juristo is a full professor at the Technical University of Madrid's School of Computing. Her research interests include empirical software engineering and software process and usability engineering. She received her PhD in computing from the Technical University of Madrid. Contact her at the School of Computing, Tech. Univ. of Madrid, Campus de Montegancedo 28660, Boadilla del Monte, Madrid, Spain; natalia@fi.upm.es.

Ana M. Moreno is an associate professor at the Technical University of Madrid's School of Computing. Her research interests are empirical software engineering, usability engineering, and conceptual modeling. She received her PhD in computing from the Technical University of Madrid. Contact her at the School of Computing, Tech. Univ. of Madrid, Campus de Montegancedo 28660, Boadilla del Monte, Madrid, Spain; ammoreno@fi.upm.es.

Sira Vegas is an assistant professor at the Technical University of Madrid's School of Computing. Her research interests focus on software testing and empirical software engineering. She received her PhD in computing from the Technical University of Madrid. Contact her at the School of Computing, Tech. Univ. of Madrid, Campus de Montegancedo 28660, Boadilla del Monte, Madrid. Spain; svegas@fi.upm.es.

Martín Solari is an assistant professor in the School of Computing at ORT University Uruguay and is researching his PhD at the Technical University of Madrid. His research focuses on software testing. Contact him at the Dept. of Software Eng., ORT Uruguay Univ., Cuareim 1451, 11100 Montevideo, Uruguay; martin.solari@ort.edu.uy.

24. G. Rothermel and M.J. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Trans. Software Eng.*, vol. 24, no. 6, 1998, pp. 401–419.
25. J. Bible, G. Rothermel, and D. Rosenblum, "A Comparative Study of Coarse—and Fine-Grained Safe Regression Test Selection," *ACM Trans. Software Eng. and Methodology*, vol. 10, no. 2, 2001, pp. 149–183.
26. F. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, 1998, pp. 44-53.
27. J.M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Application Frequency," *Proc. 22nd Int'l Conf. Software Eng.* (ICSE 00), IEEE CS Press, 2000, pp. 126–135.
28. D. Leon, W. Masri, and A. Podgurski, "An Empirical Evaluation of Test Case Filtering Techniques Based on Exercising Complex Information Flows," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), IEEE CS Press, 2005; pp. 412–421.
29. T.L. Graves et al., "An Empirical Study of Regression Test Selection Techniques," *ACM Trans. Software Eng. and Methodology*, vol. 10, no. 2, 2001, pp. 184–208.
30. G. Rothermel et al., "Test Case Prioritization: An Empirical Study," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, 1999, pp. 179–188.
31. S. Elbaum, A.G. Mailshevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," *Proc. Int'l Symp. Software Testing and Analysis*, ACM Press, 2000, pp. 102–112.
32. G. Rothermel et al., "On Test Suite Composition and Cost-Effective Regression Testing," *ACM Trans. Software Eng. and Methodology*, vol. 13, no. 3, 2004, pp. 277–331.
33. W.E. Wong et al., "A Study of Effective Regression Testing in Practice," *Proc. 8th Int'l Symp. Software Reliability Eng.*, 1998, pp. 264–274.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.