When to Test Less

Small-scale software projects usually can't afford to implement time-consuming and expensive tests. However, the authors show that in a surprisingly large number of cases, a small number of randomly selected tests will adequately probe the software.

Tim Menzies, *University of British Columbia*
Bojan Cukic, *West Virginia University*

Testing software in the large is different from testing it in the small. For large projects, we can define and develop elaborate automated *software test environments* and then apply them to the current project.[1] However, small-scale projects might not have the funds to purchase STEs or the time to build their own. In such cases, software-in-the-small projects must use manual or very simple automatic testing methods—for example, they might set input parameters to some value chosen at random between their maximum and minimum values.

In this article, we offer a justification for reducing the effort associated with software-in-the-small testing. (See the "Caveats" sidebar for information regarding the article's content and scope.) We base our argument on what we know of testing cost–benefit curves. We express *benefits* as the likelihood that a sequence of tests will reveal a fault and *costs* as the number of tests required to achieve that benefit.

After making some simple assumptions about a program's average *shape* (we define shape later in the article), we show that, usually, numerous tests probe a program no better than a small number of tests do. On average, elaborate and expensive testing regimes will not yield much more information than inexpensive manual or simple automatic testing schemes.

The Theory of Testing

In theory, software-in-the-small projects might never have the resources required to adequately test their systems. The theory of black-box testing cautions that a project could require thousands of black-box tests to determine—with only moderate confidence—that all faults have been detected within the software.

In black-box probing, we assume nothing about the program's internals. Even for systems we build ourselves, we cannot be 100% sure (due to typographical errors) of the system's internal structure.

To perform a black-box test, we randomly select test inputs from a space of plausible inputs. Each such input probes the program and might find faults. How many random black-box probes do you need to find a fault? A randomly chosen input has odds x that it will stumble across some fault. Furthermore, this input will miss that fault with odds $1 - x$. If we conduct N random black-box probes, then the odds of a failure not occurring (thus not revealing the fault) is $(1 - x)^N$. Hence, the probability y of finding a fault in N random tests is

$$y = 1 - (1 - x)^N. \quad (1)$$

Caveats

Before beginning, we pause for three caveats. First, this article presents an average-case analysis of the recommended effort associated with testing. By definition, such an average-case analysis says little about extreme cases of high criticality. Hence, our analysis must be used with care if applied to safety-critical software. On the other hand, we doubt that programming-in-the-small groups will develop such safety-critical software. Such software costs in excess of $1 million per thousand lines of code[1]—small teams probably can't develop it quickly.

Secondly, this article does not distinguish between testing conducted at different phases of the life cycle—unit testing, integration testing, product testing, or regression testing. We believe our analysis is applicable, regardless of when testing is performed.

Lastly, due to space limitations, this article skips some of the details of our mathematical model. Full details appear elsewhere.[2]

References

1. R. Schooff and Y. Haimes, "Dynamic Multistage Software Estimation," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 29, Vol. 2, 1999, pp. 272–284.
2. T. Menzies et al., "Testing Nondeterminate Systems," *ISSRE 2000*, 2000; http://tim.menzies.com/pdf/00issre.pdf (current Aug. 2000).

We usually choose x so it represents the desired software reliability. Equation 1 gives us a feel for the cost-benefit curve of black-box testing. Consider a search for a moderately low-frequency event—a one-in-10,000 event where $x = 0.0001$. Forty-six thousand randomly selected tests are required to be 99% certain that we will reveal that event.[2]

Typically, software development in the small does not use tens of thousands of tests on its software. Based on our exposure to small, commercial software development groups in the US and Australia, we claim that many programs in the small are shipped after conducting anywhere from dozens to hundreds of manually selected tests.

Black-box testing can be expensive, and other techniques—such as formal-methods testing—might not be any more promising. In formal methods, analysts specify their system's essential details and a set of logical constraints that the system must never violate. Automatic tools can then check to see if the system can violate those constraints. The benefit of formally checking a system is that such formal proofs can find faults more systematically than standard testing. A single formal first-order query is equivalent to many black-box test inputs.[3]

Unfortunately, the computational cost of such rigorous formal analysis can be impractically high. A rigorous analysis of formal properties implies a full-scale search through the system's model. This space can be too large to explore, even using today's fast machines. Hence, we can only use formal methods to test small, critical portions of our systems, and we can only justify the expense of formal methods after an extensive initial black-box phase.[3] This earlier phase serves to focus the test engineer on a program's small, highly critical portions. Thus, even when applying formal verification, we recommend first using black-box testing.

Another alternative is white-box testing, in which analysts reflect over a program's internals to invent test inputs that exercise K different partitions. Each partition exercises one interesting feature of a program—a particular bug, for example. An unexpected but repeated mathematical result is that the odds of detecting a fault with white-box methods are nearly the same as with black-box methods.[2] Even assuming certain special cases that favor white-box methods (for example, all inputs are equally likely), white-box testing using K partitions is only ever K times better at finding errors than black-box testing.[4] Building the partitions is time-consuming, so analysts might not generate many partitions. Hence, white-box testing only slightly reduces the number of tests required by black-box methods.

The Reality of Testing

The previous section provided a rather pessimistic view of software testing. However, in reality, software-in-the-small projects generate software without using thousands of tests. The software generated sometimes crashes—perhaps at the most awkward or dangerous moment—but given what we know about the mathematics of black-box testing, it's puzzling that the software doesn't crash more often.

To explain why software-in-the-small products don't crash more often, we have to consider a program's *shape*. For the moment, we'll say that a program's shape is the shape of the *pathways within the program*. If our program pathways are numerous and tangled like spaghetti, then it will be hard for the test input to navigate through the program to reach a region where we can see a bug. If our program pathways are few and not complex, then it's easy to find inputs that can reach most of the system. Having introduced the concept of shape, we can

now speculate on an answer to the question: What should the shape of software be to adequately test it using limited resources?

Numerous studies—three of which we discuss here—suggest that simple shapes are common.[5] Bieman and Schultz reported very simple shapes within a seemingly complex natural-language processing system.[6] They studied how many sets of inputs are required to exercise all *du-pathways* in a system. A du-path is a link from where a variable is defined to where it is used. Clearly, the upper bound on the number of du-pathways in a program is exponential to the number of program statements. The lower bound on the du-pathways is 1—that is, the tail of each path touches the head of another path. Note that as the du-paths shrink, the number of inputs required to reach part of the program also shrinks. Figure 1 shows their experimental results. At least for the system Bieman and Schultz studied, in the overwhelming majority of their modules, very few inputs exercised all the du-pathways.

Harrold, Jones, and Rothermel also reported simple shapes within C and Fortran functions.[7] They studied how control-flow diagrams grow as programs grow. A control-flow diagram links program statements: `While`, `Repeat`, and `For` statements introduce loops into the control-flow graph because after these statements, program control can loop back to the start of the iteration. In addition, a sequence of commands becomes a single-parent tree in the control-flow diagram. A worst-case control-flow graph is one in which every statement links to every other statement—the edges in the graph grow with the square of the number of statements. However, for over 4,000 Fortran routines and 3,147 C functions, the control-flow graph grows linearly with the number of statements. That is, at least in the system seen in Harrold, Jones, and Rothermel's study, the control-flow diagram forms almost a single-parent tree.[7] Consider the nodes 10% down such a tree. If a randomly selected input struck such nodes, then 90% of the system would be exercised.

Horgan and Mathur reported that testing often exhibits a *saturation* effect—most program paths get exercised early with little further improvement as testing continues.[8] Saturation is consistent with programs containing either many portions with simple shapes—so the portions are easily reached—or many portions that are so twisted in shape that that we'll never reach them.

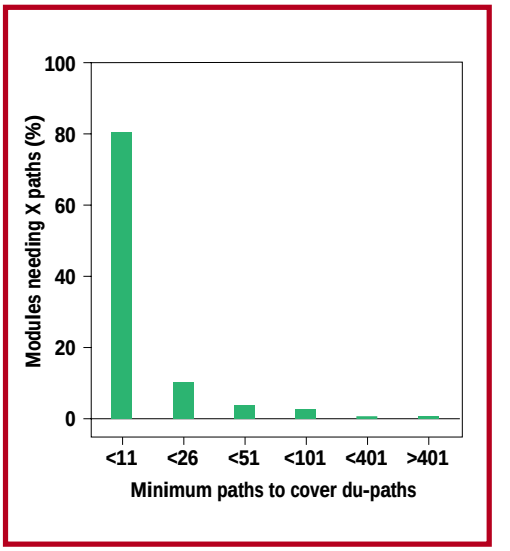

Figure 1. An analysis of hundreds of modules in a software system.

If these three examples represent a general case, then we can avoid the pessimistic conclusions of the theory of black-box testing.

The Average Shape of Software

If we can show that our software usually has the *right shape*, then we can declare that we can usually test it quickly. By right shape, we mean that either the program's shape is so complex that no amount of testing will reach regions where the program will fail or that the program's shape is so simple that a few randomly selected tests will reach regions where the program will fail. In either case, there is no point in conducting lengthy and expensive testing because a limited testing regime will yield as much information as an elaborate testing procedure.

What then is the average shape of our software—is it the right shape for easy testing? To find out, we constructed a mathematical model of building a pathway across a program from randomly selected inputs to some randomly selected state. By simulating the model for a wide range of parameters, we can infer that, on average, our programs are indeed the right shape for simple testing.

To implement the model, we make some assumptions about the tested program's structure.

Assumption 1

Programs are networks connecting system concepts. For example, Figure 2 shows a network of concepts inside a program. (We explain the use of `no`, `yes`, `or`, and `and` later).

This first assumption is hardly controversial. Any optimizing compiler builds a network (control and dataflow graphs) from the code. Optimization is then a matter of reorganizing the network to speed up the pro-

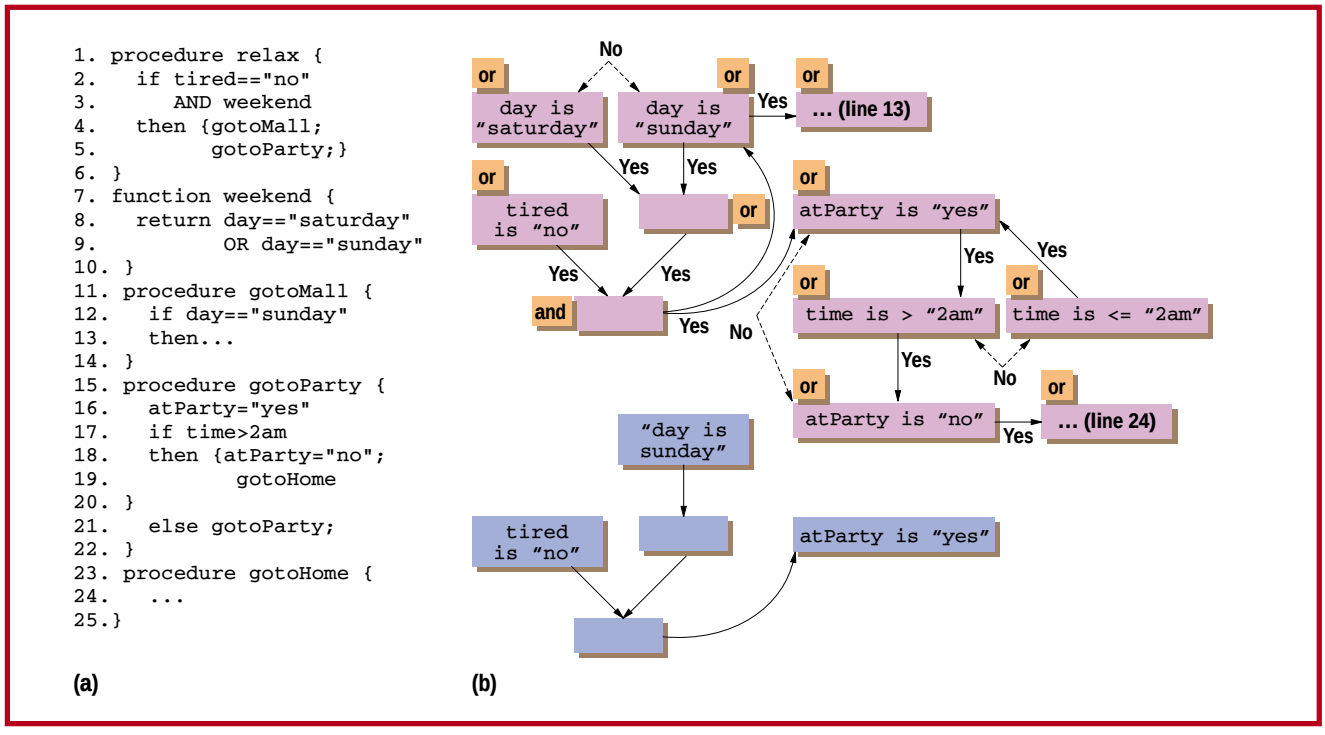

Figure 2. (a) Conversion of procedural code to (b) a NAYO (no-and-yes-or) network. The blue shaded tree is an explanation for how we might use this system to get to a system bug (in this case, letting our programmers get to a party).

gram. When a program executes, it starts at the inputs and then runs over that network. Again, it is not controversial to view execution as the generation of trees from the program:

- An execution trace of a procedural program shows what statements are executed, which subroutines are called, and in what order.
- A general method for reasoning about programs is to create a set of axioms describing, for example, preconditions and postconditions. Any theorem proof procedure over those axioms builds a proof tree. Such a proof tree would qualify as our execution tree.

We view testing as just a special case of execution in which we record the tree of pathways followed over a network and terminate the execution of testing when we arrive at something interesting (a fault, for example).

Assumption 2

A fault explanation tree *is a tree whose leaves are inputs and whose root is a fault.*

Referring to Figure 2, suppose that we never want our programmers to go to parties. If we could generate a tree that arrives at `atParty="yes"`, then we would have detected a fault. The shaded tree at the bottom of Figure 2b shows that, indeed, our programmers can get to parties—in other words, there is a bug in our system.

Assumption 3

Generalizing the example in Figure 2, we assume that *testing is a process of trying to generate a fault explanation tree from the program network.*

If we can't generate such an explanation, we gain confidence that there are no faults in our program.

Assumption 4

An explanation tree has edges and nodes. There are two kinds of nodes: `and` nodes and `or` nodes. There are two kinds of edges: `yes` edges and `no` edges.

`And` nodes model conditional statements. For example, consider lines 2 and 3 of the code shown in Figure 2. This conjunction generates one `and` node with two parents: one for `tired="no"` and the other for the test for `weekend`. Only if all the parents of an `and` node are satisfied do we move to the `then` part of the conditional statement (lines 4 and 5).

`Or` nodes model the disjunction of state-

Figure 3. An average case of testability. For simplicity's sake, this description does not include certain details of the full version of the model. For example, this simple description makes no reference to NAYO graphs with nonuniform structure, loop detection, or contradiction detection. For full details, see the full description.

ments. For example, the statement `day=="saturday" OR day=="sunday"` generates an `or` node with two parents. If any of the parents of an `or` node are satisfied, then we move to the `then` part of the conditional.

`No` edges model what can't be believed at the same time. For example, we can't believe that `day=="saturday"` and `day=="sunday"`. Hence, we connect these nodes with a `no` edge. `No` edges only connect `or` nodes because `and` nodes have no inconsistencies.

`Yes` edges denote where we can move through the program (providing that our trace does not include any pair of nodes connected by `no` edges). For example, `yes` edges connect the parents and children of our `and` nodes and `or` nodes.

We call a network of `no`, `and`, `yes`, and `or` components a NAYO network. Now we can provide a precise definition of testing.

Assumption 5

Testing is the process of extracting NAYO trees (the explanations) from NAYO networks (the program).

Implementation

These assumptions are sufficient to simulate the construction of an explanation tree across a program (see Figure 3). We can approximately characterize the tree's shape by the number of tests N required to reach that tree's root to find a fault. Trees with an overly complex shape require an impractically large N to reach their root, while trees with a simple shape require a very small N (the number of tests) to reach their root. More precisely,

$$Shape = \begin{cases} simple\ if & 0 < N < 10^2 \\ moderate\ if & 10^2 \le N < 10^4 \\ hard\ if & 10^4 \le N < 10^6 \\ overly\ complex\ if & N \ge 10^6 \end{cases}$$

We simulated this model 150,000 times for a wide range of the parameters:

- up to $V = 10^8$ nodes,
- up to $in = 1{,}000$ inputs, and
- wildly varying frequency of `and` nodes `andf`, `or` nodes `orf`, `and` parents `andp`, and `or` parents `orp`.

Table 1 shows the frequency distribution of the N values calculated from the NAYO model. We made two important observations from this frequency distribution. The first is the 56% observation (which combines the 36% *simple* shapes with the 20% *overly complex* shapes): over half the simulations found very simple or overly complex shapes. Thus, over half the time, 100 random tests will yield as much information as a much more prolonged series of tests (up to a million tests). The second observation is the 75% observation (which combines the results seen for simple (36%), moderate

---

Given *in* inputs to a NAYO network with V nodes, the odds of hitting a fault straight away from the inputs is

$$x_0 = \frac{in}{V} \quad . \tag{A}$$

The probability of reaching an `and` node with `andp` parents is the probability of reaching all its parents:

$$x_{and} = x_i^{andp} \tag{B}$$

where x_i is the probability we computed in the prior step of the simulation (and the base case of $x_{i=0}$ is computed from Equation A).

The probability of reaching an `or` node with `orp` parents is the probability of not missing any of its parents; that is,

$$x_{or} = 1 - (1 - x_i)^{orp}. \tag{C}$$

If the ratio of and nodes in a NAYO network is `andf`, then the ratio of `or` nodes in the same network is `or f = 1 - andf`. The odds of reaching some random node x_j is the weighted sum of the probabilities of reaching `and` nodes or `or` nodes:

$$x_j = andf * x_{and} + orf * x_{or} \ . \tag{D}$$

We can rearrange Equation 1 from the main text to isolate the number of tests required to be 99% sure of finding a fault with probability x_j:

$$y = 0.99 = 1 - ((1 - x_j)^N)$$
$$\therefore N = \frac{\log(1 - 0.99)}{\log(1 - x_j)} \tag{E}$$

(19%), and overly complex (20%) shapes): in 75% of the simulations, 10,000 randomly selected tests will probe the program as much as a very prolonged series of tests (up to a million tests) will.

For software-in-the-small projects, a cost-cutting heuristic might be to avoid elaborate and expensive testing regimes. The mathematics of black-box testing (see Equation 1) pessimistically concludes that this is a dangerous heuristic. According to that math, the adequate testing of programs requires a prolonged series of tests.

Black-box mathematics is blind to the internal structures and shapes of our program. If we include internal structure in our analysis, the pessimistic conclusion of black-box testing disappears. Furthermore, our average case analysis of the shape of our programs strongly suggests that simple and inexpensive automatic random testing methods can be quite valuable.

Based on this analysis, we advise that software-in-the-small projects should routinely execute overnight test runs in which tens of thousands of test cases are generated at random from the known legal ranges of system input. It is important to stress that these overnight testing runs must be truly random. Biases in the selection of input data would violate the assumption of random search used in our NAYO model. Elsewhere, we describe general principles for managing really randomized probing.[10,11]

Our analysis describes general conclusions for classes of programs (the simple-shaped programs, the hard-shaped programs, and so forth). Our current research goal is to apply this analysis to some particular program. To this end, we are extending static code analysis tools to extract the parameters seen in our mathematical model from real-world programs. Once we can do that, we can monitor the evolution of a particular program and detect whether that program has, for example, suddenly veered away from simple, testable shapes. Furthermore, we hope to offer design guidelines such that analysts can design their systems to avoid hard-shaped programs that require elaborate and expensive test regimes.

About the Authors

Tim Menzies is an assistant professor at the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada. He is a cognitive scientist exploring how quirks in human cognition affect the process of software and knowledge engineering. He holds a PhD in artificial intelligence, an MS in cognitive science, and a BS in computer science—all from the University of New South Wales, Sydney, Australia. He is the cofounder and organizer of WISE: the International Workshop on Intelligent Software Engineering, and he was chair of WISE 2000 at ICSE 2000. Contact him at tim@menzies.com; http://tim.menzies.com.

Bojan Cukic is an assistant professor at the Department of Computer Science and Electrical Engineering, West Virginia University. He recently became a codirector of the NASA/WVU Software Research Laboratory at NASA IV&V Facility in Fairmont, WV, where he leads research efforts in novel testing methodologies, software reliability assessment for safety-critical applications, and software certification techniques for intelligent flight control systems. His research interests include software engineering, high-assurance systems, and high-performance computing. He received his Dipl. Ing. degree from the University of Ljubljana, Slovenia, and his MS and PhD in computer science from the University of Houston. In 1998, he received the Outstanding Young Researcher Award in the College of Engineering and Mineral Resources at West Virginia University. Contact him at cukic@csee.wvu.edu.

Table 1

Results from the Simulation Runs

Classification	Threshold	Percent
Simple	$0 < N < 10^2$	36
Moderate	$10^2 \le N < 10^4$	19
Hard	$10^4 \le N < 10^6$	25
Overly complex	$N \ge 10^6$	20

Acknowledgments

NASA partially supported this work through cooperative agreement No. NCC 2-979.

References

1. N. Eickelmann and D. Richardson, "An Evaluation of Software Test Environment Architectures," *Proc. Int'l Conf. Software Eng.*, Springer-Verlag, Berlin, 1996, pp. 353–364.
2. D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, Vol. 16, No. 12, Dec. 1990, pp. 1402–1411.
3. M. Lowrey, M. Boyd, and D. Kulkarni, "Towards a Theory for Integration of Mathematical Verification and Empirical Testing," *Proc. ASE'98: Automated Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998, pp. 322–331.
4. W. Gutjhar, "Partition vs. Random Testing: The Influence of Uncertainty," *IEEE Trans. Software Eng.*, Vol. 25, No. 5, 1999, pp. 661–674.
5. T. Menzies and B. Cukic, "Adequacy of Limited Testing for Knowledge Based Systems," to be published in *Int'l J. Artificial Intelligence Tools (IJAIT)*, 2000; http://tim.menzies.com/pdf/00ijait.pdf (current Aug. 2000).
6. J. Bieman and J. Schultz, "An Empirical Evaluation (and Specification) of the All-Du-Paths Testing Criterion," *Software Eng. J.*, Vol. 7, No. 1, 1992, pp. 43–51.
7. M. Harrold, J. Jones, and G. Rothermel, "Empirical Studies of Control Dependence Graph Size for C Programs," *Empirical Software Eng.*, Vol. 3, 1998, pp. 203–211.
8. J. Horgan and A. Mathur, "Software Testing and Reliability," *The Handbook of Software Reliability Eng.*, M.R. Lyu, ed., McGraw-Hill, New York, 1996, pp. 531–565.
9. T. Menzies et al., "Testing Nondeterminate Systems," to be published in *ISSRE 2000: Int'l Symp. Software Reliability Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 2000; http://tim.menzies.com/pdf/00issre.pdf (current Aug. 2000).
10. T. Menzies and E. Sinsel, "Practical Large Scale What-If Queries: Case Studies with Software Risk Assessment," *Proc. IEEE Int'l Conf. Automated Software Eng. 2000*, IEEE Computer Soc. Press, Los Alamitos, Calif., 2000; http://tim.menzies.com/pdf/00ase.pdf (current Aug. 2000).
11. T. Menzies and C. Michael, "Fewer Slices of PIE: Optimizing Mutation Testing via Abduction," *Int'l Conf. Software Eng. and Knowledge Eng. '99*, World Scientific, New Brighton, Minn., 1999; http://tim.menzies.com/pdf/99seke.pdf (current Aug. 2000).