COVER FEATURE

# The Strangest Thing About Software

*Tim Menzies and David Owen,* West Virginia University
*Julian Richardson,* Research Institute for Advanced Computer Science

**Although there are times when random search is dangerous and should be avoided, software analysis should start with random methods because they are so cheap, moving to the more complex methods only when random methods fail.**

The physicist John Archibald Wheeler advised that "in any field, find the strangest thing and explore it." Accordingly, we explore the strangest thing about software—that it ever works at all.

Modern software is so complex that it should never work. For example, we once wrote a search engine that tried to find all the unreachable goals in a model (http://menzies.us/pdf/95thesis.pdf). Our first implementation was impractically slow and a little mathematics showed us why: We were searching a model with 300 Boolean variables. Such a model has up to $2^{300} = 2 \times 10^{90}$ different states. To put that number in perspective, astronomers estimate that the universe holds $10^{24}$ stars. That is, this little model had more internal states than stars in the sky.

It is impossible to rigorously search software implementations of such complex models, as the "Exponential Cost of Traditional Assessment" sidebar shows. Any software assessment or verification and validation process must negotiate this complexity by effectively covering only a fraction of the software's possible internal configurations.

So why does software work? Why aren't our incomplete test methods missing all too many critical errors? One response might be to deny the premise and argue that software rarely works as well as it should. To be sure, software sometimes crashes—perhaps at the most awkward or dangerous moment. See, for example, the depressing litany of mistakes documented in Peter Neumann's *Computer-Related Risks.*[1] However, given software's internal complexities, that it doesn't crash routinely is very strange.

We propose that software works because internally it is surprisingly simple. This proposal is based on recent results from artificial intelligence research. AI has discovered certain previously unrecognized regularities that developers can use to quickly find solutions. A careful reading of the software engineering literature shows that these regularities have also been seen in conventional software. For problems with those regularities, much of what we can find via complex and costly methods, we can also find by random search. This is an important result because an incomplete randomized algorithm might be the simplest one available, the fastest, or both.[2]

## COLLARS

Because AI has historically endured a bad reputation, it might seem strange to assert that AI can help software analysis. To explain this assertion, we begin with a review of some interesting results from that field.

AI researchers have repeatedly concluded that a small number of key variables determine the behavior of the rest of the system. We call these *collar*[3] variables. When collars are present, the problem of controlling software reduces to just the problem of controlling the variables in the collar. Consequently, a developer searching for bugs only needs to test just enough to sample the range of the collar variables (and sampling the rest of the code is far less cost-effective since it has much less influence on program behavior). Collars have been discovered and rediscovered in AI many times and given different names,

Published by the IEEE Computer Society 0018-9162/07/$20.00 © 2007 IEEE

## Exponential Cost of Traditional Assessment

If $x$ is the probability that a randomly selected program input finds a fault, then after $N$ random inputs, the chance of the inputs not revealing a fault is $(1 - x)^N$. Hence, the chance $C$ of seeing the fault is $1 - (1 - x)^N$, which can be rearranged to $N = log(1 - C)/log(1 - x)$. This expression shows that a linear increase in $C$ requires exponentially more tests. For example, for one-in-a-thousand detects ($x = 0.001$), moving $C$ from 90 to 94 to 98 percent requires 2,301, 2,812, and 3,910 black-box probes, respectively.[1]

The problem of requiring exponential resources to evaluate software cannot be solved using more informed evaluation methods. For example, the infamous state-space explosion problem imposes strict limits on how much a system can be explored via, say, automatic formal methods.[2] With Bojan Cukic, we have explored many other examples in which assessment effectiveness is exponential on effort.[3]

### References

1. J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software*, May 1995, pp. 17-28.
2. M. Lowry, M. Boyd, and D. Kulkarni, "Towards a Theory for Integration of Mathematical Verification and Empirical Testing," *Proc. Automated Software Eng.* (ASE 98), 1998, pp. 322-331.
3. T.J. Menzies and B. Cukic, "How Many Tests Are Enough?"; http://menzies.us/pdf/00ntests.pdf.

including *variable subset selection*,[4] *narrows*,[5] *master variables*,[6] and *back doors*.[7] For example, many researchers have examined what happens when a data miner deliberately ignores some of the variables in the training data. In one study, Ron Kohavi and George John studied a specific variable subset selection method.[4] Their experiments on eight real-world data sets show that an average 81 percent of variables can be ignored. Further, ignoring those variables doesn't degrade the learner's classification accuracy. On the contrary, it results in an average increase in accuracy, as Table 1 shows.

Saul Amarel observed that search problems contain tiny collars—which he called *narrows*—in their search space that must be traversed in any solution.[5] In such a space, what matters is not so much how we get to these collars, but what decision we make when we get there. Since the route between collars is not important, Amarel's work defined macros that encode paths between them in the search space, effectively permitting a search engine to jump between them. Amarel's narrows would explain the variable subset selection results: We can ignore variables from outside the narrows without losing control of a system.

James Crawford and Andrew Baker observed collars, which they called *master variables*, while investigating different scheduling methods.[6] They built Tableau, a very fast, complete search engine and compared its performance to Isamp, a simple randomized search engine. Both algorithms assign a value to one variable, then infer some consequences with forward checking. If the algorithms detect contradictions, Tableau backtracks while Isamp simply starts over and reassigns other variables randomly, giving up after `MAX-TRIES` number of times. Otherwise, as Figure 1 shows, both algorithms continue looping until all variables are assigned. Surprisingly, Isamp took less time than Tableau to find more scheduling solutions using just a small number of `TRIES`.

Crawford and Baker explained this effect by assuming that a small set of master variables set the remaining variables in a system. They hypothesized that the solutions are not uniformly distributed throughout the search space. Tableau's depth-first search sometimes wanders into regions containing no solutions by making an early unlucky choice in the master variables. On the other hand, Isamp's randomized sampling effectively searches in a smaller space because it restarts on every contradiction.

Crawford and Baker argued that the collars play an important role in controlling how long it takes to find a solution. A similar conclusion comes from the work of Ryan Williams, Carla Gomes, and Bart Selman, who discuss how to use collars—which they call *back doors*—to optimize search.[7] Constraining the collars constrains the rest of the program as well, by definition. So, to quickly search a program, they suggest imposing some setting on the collar variables. This reduces the remaining search space within a program, which can then be explored quickly. Some researchers argue that

**Table 1. Variable subset selection results.[4]**

| | Average number of variables | | Accuracy change | |
|---|---|---|---|---|
| **Data set** | **Before** | **After** | **After (percentage)** | **Before (percentage)** |
| Breast cancer | 10 | 2.9 | 29 | +0.14 |
| Cleve | 13 | 2.6 | 2 | +5.89 |
| Crx | 15 | 2.9 | 19 | +4.49 |
| DNA | 180 | 11.0 | 6 | +3.63 |
| Horse colic | 22 | 2.8 | 13 | +1.63 |
| Pima | 8 | 1.0 | 13 | +0.79 |
| Sick-euthyroid | 25 | 4.0 | 16 | +0.38 |
| Soybean | 35 | 12.7 | 36 | +0.15 |
| **Average** | 38.5 | 4.99 | 19 | +2.14 |

```
for i := 1 to MAX TRIES {
   try:
    set all variables to unassigned
    loop {
       if all variables are valued
       then return current assignment
       else{ v ← random unvalued variable
             assign v a randomly chosen value
             unit_propagate()
             if contradiction goto try
   }}}
return failure
```

| | Tableau: Full search | | Isamp: Partial, random search | | |
|---|---|---|---|---|---|
| | Success rate | Time in seconds | Success rate (percent) | Time in seconds | Tries |
| A | 90% | 255.4 | 100 | 10 | 7 |
| B | 100% | 104.8 | 100 | 13 | 15 |
| C | 70% | 79.2 | 100 | 11 | 13 |
| D | 100% | 90.6 | 100 | 21 | 45 |
| E | 80% | 66.3 | 100 | 19 | 52 |
| F | 100% | 81.7 | 100 | 68 | 252 |

***Figure 1. Average performance of Tableau versus Isamp on six scheduling problems, A through F, with different levels of constraints and bottlenecks.[6] Isamp's `unit_propagation` procedure is a special linear-time case of resolution.***

this policy can reduce exponential time problems to polynomial time—providing that it's possible to cheaply locate the collars.[7]

## EVIDENCE FROM SOFTWARE ENGINEERING

If collars are present in conventional—that is, non-AI software—then a few collar variables determine a software package's overall behavior. If so, then we would expect three effects:

- Software testing should quickly *saturate*—most program paths will be exercised early, with little further improvement seen as testing continues.
- Random *mutation* will be more likely to affect the many non-collar variables compared to the few collar variables, so the net effect of those mutations would be small (a *mutant* of a program is a syntactically valid but randomly selected variation to a program, such as swapping all plus signs to a minus sign). If so, then most random mutations of a program containing collars will not change the program's behavior.
- Software states should *clump* so that only a small number of states will be reached at runtime. Collars imply clumps because the number of reachable states in an application will be quite small, containing just the number of possible settings to the collar.

Actually, clumps can also cause collars. Collars store the differences between the states reached at runtime. If the number of states is small, the number of differences will also be small.

All these effects can be found in the software engineering literature. Joseph Horgan and Aditya Mathur reported the saturation effect.[8] As to mutation testing, Christoph Michael found that in 80 to 90 percent of cases, there were no changes in the behavior of a range of programs despite numerous perturbations on data values using a program mutator.[9] In similar results, Eric Wong compared results using $X$ percent of a mutator library, randomly selected ($X \in \{10\%, 15\%, \ldots 40\%, 100\%\}$).[10] Most of what could be learned from the program could be learned using only $X = 10$ percent of the

## Expected Distribution of Reachable States in Software

If software has $n$ variables, each with its own assignment probability distribution of $p_i$, then the probability that software will fall into a particular state is

$$p = p_1 p_2 p_3 \qquad p_n = \prod_{i=1}^{n} p_i .$$

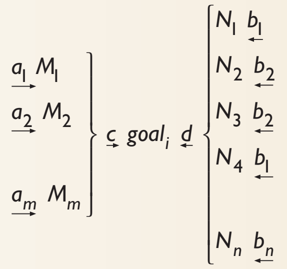
By taking logs of both sides, this equation becomes

$$\ln p = \ln = \prod_{i=1}^{n} p_i = \sum_{i=1}^{n} \ln p_i$$

The central limit theorem addresses the asymptotic behavior of such a sum of random variables. In the case where we know little about software, $p_i$ is uniform and many states are possible. However, the more we know about software, the more varied are individual distributions. Given enough variance in the individual priors and conditional probabilities, or $p_i$, we expect that the frequency with which we reach states will exhibit a lognormal distribution, in which we can expect a small fraction of states to cover a large portion of the total probability space, and the remaining states have practically negligible probability.

## Likelihood of Clumps

Suppose the output space of software has been bunched together into a small number of equivalence classes (for example, numerous numeric outputs all scored "low number of errors"). In such software, there are many ways to reach equivalent output. Consider the space of possible inference chains within such software. Some of these intersect and can clash over the value of a variable at the intersection. We say that collars contain the clashes that were not dependent on any other clashes. Let some goal in software be reachable by a narrow collar *M* or a wide collar *N*:



Let the cardinality of the narrow funnel and wide funnels be *m* and *n* respectively. Each *m* member of *M* is reached via a path with probability $a_i$, while each *n* member of *N* is reached via a path with probability $b_i$. Two paths exist from the funnels to this goal: one from the narrow neck with probability *c* and one from the wide neck with probability *d*.

The probability of reaching the goal via the narrow pathway is

$$narrow = c\prod_{i=1}^{m} a_i$$

while the probability of reaching the goal via the wide pathway is

$$wide = d\prod_{i=1}^{n} b_i$$

For what values of *m* and *n* are the odds $narrow >> wide$? In the case of uniform distributions of $a_i$, $b_i$ where

$$\sum_{i=1}^{m} a_i = 1,\ \sum_{i=1}^{n} b_i = 1,\ a_i = \frac{1}{m},\ b_i = \frac{1}{n}$$

then we have shown that at, for example, $m = 3$, the wider collar pathway is very unlikely. Precisely, the wider collar pathway is favored when

$$\frac{d}{c} \geq 1{,}728,$$

that is, only in the unlikely case that the *d* pathway is thousands of times more likely than *c*.

We have built a small simulator to study the nonuniform case and reached the same conclusions: narrow collars were millions of times more likely.[1]

**Reference**

1. T. Menzies and H. Singh, "Many Maybes Mean (Mostly) the Same Thing"; http://menzies.us/pdf/03maybe.pdf.

mutators. After a very small number of mutators, new mutators acted the same as previously used ones. Timothy Budd[11] and Allen Acree[12] also have made this observation.

Marek Druzdzel (www.pitt.edu/~druzdzel/abstracts/uai94.html) observed clumping in a diagnosis application for monitoring patients in intensive care. Although the software had 525,312 possible internal states, the application reached few of them at runtime: One of the states occurred 52 percent of the time, and 49 states appeared 91 percent of the time. Druzdzel could show mathematically that there is nothing unusual about his application. Thus, we should always expect that software will clump, as the "Expected Distribution of Reachable States in Software" sidebar describes.

Empirical evidence for clumping also comes from Radek Pelanek's detailed review of the structures of dozens of formal models.[13] He found that, on average, their internal structure was remarkably simple. Formal models often comprised one large, strongly connected component (where if state *u* connects to state *v*, *v* also connects to *u*) and small *diameters* (the largest, shortest path between two states was quite short). A program executing around such a space would repeatedly arrive back at a small number of states, thus clumping.

## USING COLLARS AND CLUMPS

To exploit collars and clumps, we must first note that they dramatically reduce the search space within a program. The number of variables in a collar should be very small, as the "Likelihood of Clumps" sidebar explains. The collar variables set the number of reachable states, so small collars also mean that the number of clumping states will be small. Hence, theoretically, random search will quickly find most of what can be found via a more complete search.

We have been testing this theoretical speculation since 1999 (http://menzies.us/pdf/99seke.pdf). Currently, we are experimenting with two random search algorithms that show much promise: LURCH and TAR3.

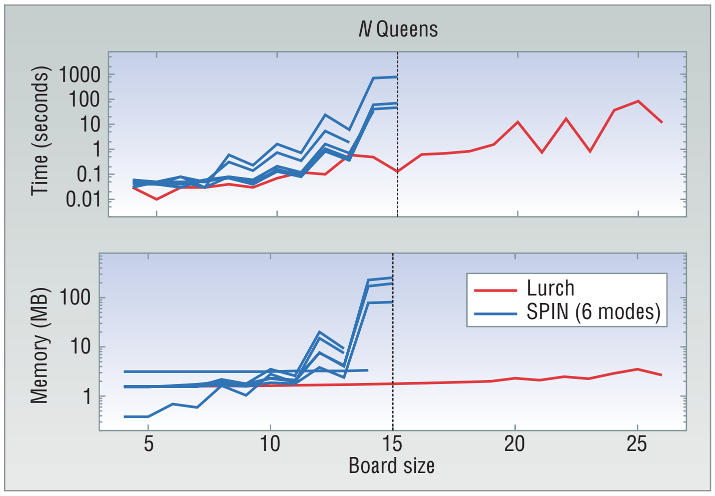


***Figure 2. Solving the N-queens problem using complete (SPIN) versus random (LURCH) search. SPIN was run in six modes, each using a different set of efficiency-improving options. The one horizontal line on the memory graph for SPIN, running up to only board size 14, was generated using SPIN's lossy supertrace compression option. This saves a great deal of memory in many cases but was not accurate for models larger than board size 14.***

## TAR3

A randomized version of the TAR2 data miner[14] (http://unbox.org/wisp/tags/tar/3.0), TAR3 inputs a set of scored examples and outputs contrast rules that distinguish highly scored examples from the others. The rule generation algorithm seeks the *smallest* set of rules that most select for the *highest* scoring examples.

To find the collar variables, TAR3 assumes that if collars exist, they control software's behavior. So, a random selection of the software behaviors must, by definition, sample the collars. That is, we need not search for the collars—they'll find us.

If we generate scenarios at random, such as Monte Carlo simulations, then score each run as good or bad using some domain knowledge such as number of goals reached, the collar variables will be those with attribute ranges that occur with very different frequencies in good rather than bad runs. TAR3 builds its rules randomly, favoring attribute ranges that occur more in high-scoring examples than in lower-scoring ones.

Dustin Geletko (http://menzies.us/pdf/03radar.pdf) applied TAR3 to a version of the Limits to Growth[15] model. This model studies the effects of the world's exponentially growing population and economy. The full model contains 295 variables and more than 100 nodes. MIT faculty studied this model for several years to find factors that prevented global population overshoot and collapse, such as *desired completed family size normal* = 0..2 and *industrial capital output ratio* = 3.5.

To check TAR3's conclusions, Geletko conducted another simulation that constrained the inputs according to TAR3's recommendations. Because TAR3 returns the collar variables, and we expect that number to be very small, TAR3's learned theories should be more succinct than standard learners. To test this, Geletko gave an entropy decision tree learner the same data that TAR3 used. That learner returned a decision tree with 200 tests. In a result consistent with the collar hypothesis, TAR3's theory, learned from the same data, needed to test only two variables: *desired completed family size normal* and *industrial capital output ratio*.

The study's two main effects were that TAR3 returned very small theories and that these theories proved effective in changing the distribution of some system. Ying Hu[16] describes many studies with the algorithm and multiple data sets from the standard University of California, Irvine data mining data sets,[17] plus some software engineering domains. The two effects seen in the Limits to Growth Study also appeared in the UCI data. TAR3 always produced theories that tested less than five variables, and those theories, when applied as a SELECT statement to the data sets, selected examples with a greatly changed class distribution.

## LURCH

A random testing and debugging tool for finite-state models[18] (http://unbox.org/wisp/tags/lurch/1.0), LURCH works by simulating the execution of the models—choosing randomly when more than one transition is possible. LURCH never backtracks; it simply runs until some termination condition (such as path end, depth limit, or error detected), and then starts over until a user-specified number of paths have been explored.

For example, Figure 2 compares the times and memory required for a complete search and randomized search to solve the "*N*-queens" problem: Place *N* queens on an $N \times N$ chess board such that no queen can take any other. For the complete search, we ran the SPIN model checker[19] in six modes, each using a different set of options to improve that search. For the random search, we used LURCH to explore a finite-state model of *N*-queens.

As Crawford and Baker noted, a complete search looks into everything and can get stuck in some irrelevant corner of the problem. This effect can be seen in Figure 2: SPIN's complete search gave up and died on anything larger than a $15 \times 15$ board, shown by the vertical dashed line. Just like Isamp, however, LURCH's random search has a built-in get-out-of-jail-free card: When it gets stuck, it can jump over a wall and start afresh somewhere else. Observe how LURCH scaled to much larger problems than SPIN.

Figure 2 also demonstrates the *order effects* that plague deterministic search. For deterministic algorithms, certain inputs always result in the slowest runtimes. For example, insertion sort runs slowest if the inputs are already sorted in reverse order. In Figure 2, to cite another example, SPIN's complete search takes exponentially less time and memory for boards of odd size than for those of even size. The random search, on the other hand, jumps around the input data, so it can be difficult to find inputs that generate worst-case runtimes.

A standard objection to using incomplete random methods like LURCH for real-world problems is that they can miss important behavior or critical errors in the model. While certainly true, this objection assumes that the model is small enough to be processed by complete search. For larger models, random search may be the only viable option. Also, our recent work with Dejan Desovski and Bojan Cukic shows how random search can be used to complement complete search to produce a more reliable verification result.[18] That study used SPIN and LURCH to explore an error-seeded formal requirements model. SPIN found errors not detected by LURCH, which makes sense since SPIN carries out a complete search. Surprisingly, LURCH found one error not found by SPIN. We eventually found the cause of this strange result—the automatic translation tool used to generate the SPIN model. It implemented a memory saving with an inappropriate assumption that portions of the model were deterministic. This hid from SPIN some of the model's behavior, including the error found by LURCH.

To be sure, there are times when random search is dangerous and should be avoided. For example, the software controller of a manned spacecraft's ascent stage should be a deterministic algorithm with guaranteed performance properties. Using random search at this stage of the mission is as crazy as not using random search to assist in onboard diagnosis when the craft is in deep space, in deep trouble, all other methods are failing, and it takes too long to ask for help from ground control.

Complete methods and random methods should be mixed and matched. For the software's mission-critical kernel, tools like SPIN should be used for complete validation. But the rest of the software could be too big for complete analysis, in which case a random search using, for example, LURCH or TAR3, might be the only cost-effective option.

We recommend starting with random software analysis because it is so cheap, moving to the more complex methods only when random methods fail. Writing nearly two decades ago, Barry Boehm made an analogous proposal for iterative software exploration.[20] Writing at the same time, Donald Norman argued that such iterative exploration is essential in any human design process.[21]

Much has changed since 1988. We now know how to write algorithms that exploit certain regularities internal to software. LURCH can quickly sample the reachable clumps, and TAR3 can find the smallest rules that most select for them. This kind of randomized search and learning shows great promise for finding the key decisions within seemingly intricate software. ■

## Acknowledgments

This work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility and conducted at the West Virginia University and at the NASA Ames Research Center. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or NASA.

## References

1. P.G. Neumann, *Computer-Related Risks*, ACM Press/Addison Wesley, 1995.
2. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge Univ. Press, 1995.
3. T. Menzies and J. Richardson, "Making Sense of Requirements, Sooner," *Computer*, Oct. 2006, pp. 112-114; http://menzies.us/pdf/06qrre.pdf.
4. R. Kohavi and G.H. John, "Wrappers for Feature Subset Selection," *Artificial Intelligence*, vol. 97, nos. 1-2, 1997, pp. 273-324.
5. S. Amarel, "Program Synthesis as a Theory Formation Task: Problem Representations and Solution Methods," *Machine Learning: An Artificial Intelligence Approach*: *Volume II*, R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, eds., Morgan Kaufmann, 1986, pp. 499-569.
6. J. Crawford and A. Baker, "Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems," *Proc. American Assoc. Artificial Intelligence* (AAAI 94), AAAI Press/MIT Press, 1994, pp. 1092-1097.
7. R. Williams, C.P. Gomes, and B. Selman, "Backdoors to Typical Case Complexity," *Proc. IJCAI* 2003; www.cs.cornell.edu/gomes/papers/backdoors.pdf.
8. J. Horgan and A. Mathur, "Software Testing and Reliability," *The Handbook of Software Reliability Engineering*, M.R. Lyu, ed., McGraw-Hill, 1996, pp. 531-565.
9. C.C. Michael, "On the Uniformity of Error Propagation in Software," *Proc. 12th Ann. Conf. Computer Assurance* (COMPASS 97), 1997, pp. 68-76.
10. W.E. Wong and A.P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *J. Systems and Software*, vol. 31, no. 3, 1995, pp. 185-196.
11. T.A. Budd, "Mutation Analysis of Programs Test Data," doctoral dissertation, Yale Univ., 1980.
12. A.T. Acree, "On Mutations," doctoral dissertation, School of Information and Computer Science, Georgia Inst. of Technology, 1980.

13. R. Pelanek, "Typical Structural Properties of State Spaces," *Proc. SPIN 04 Workshop*, 2004.
14. T. Menzies and Y. Hu, "Data Mining for Very Busy People," *Computer*, Nov. 2003, pp. 22-29.
15. D.H. Meadows et al., *The Limits to Growth*, Potomac Associates, 1972.
16. Y. Hu, "Treatment Learning: Implementation and Application," master's thesis, Dept. Electrical Eng., Univ. of British Columbia, 2003.
17. C.L. Blake and C.J. Merz, *UCI Repository of Machine Learning Databases*, 1998; www.ics.uci.edu/~mlearn/MLRepository.html.
18. D. Owen, D. Desovski, and B. Cukic, "Effectively Combining Software Verification Strategies: Understanding Different Assumptions," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE 06)*, IEEE Press, 2006.
19. G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, 1997, pp. 279-295.
20. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Software Eng. Notes*, vol. 11, no. 4, 1986, pp. 61-72.
21. D.A. Norman, *The Design of Everyday Things*, Doubleday Currency, 1989.

*Tim Menzies is an associate professor at the Lane Department of Computer Science and Electrical Engineering, West Virginia University. His main research interests are data mining for software engineering. Menzies received a PhD in artificial intelligence from the University of New South Wales, Sydney, Australia. He is a member of the IEEE. Contact him at tim@menzies.us.*

*David Owen is a PhD student at the Lane Department of Computer Science and Electrical Engineering, West Virginia University. His research interests include model checking and randomized algorithms. Owen received an MS from West Virginia University. He is a student member if the IEEE. Contact him at owen@csee.wvu.edu.*

*Julian Richardson is a research scientist working for RIACS in the Reliable Software Engineering Group, NASA Ames Research Center. His main research interests include software risk assessment, verification and validation, and automated software engineering. He received a PhD in artificial intelligence from the University of Edinburgh, Scotland. He is a member of the ACM. Contact him at julian.richardson@gmail.com.*