The Strangest Thing About Software *†

Tim Menzies, West Virginia University, `tim@menzies.us`
Julian Richardson, Research Institute for Advanced Computer Science, `julianr@riacs.edu`
David Owen, Prologic Inc., `owen@freeshell.org`

Abstract

AI research explains the strangest feature of software and tell us how to understand larger programs.

1 Introduction

The physicist John Archibald Wheeler advised that "in any field, find the strangest thing and explore it." Accordingly, this article explores the most strangest thing about software; i.e. *that it ever works at all.*

Software should not work. It is too complex for us to understand it. For example, once we wrote a search engine to find unreachable goals in a model [1]. The 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, it is estimated that there are 10^{24} stars in the universe. That is, our little model had more internal states than stars in the sky.

Software implementations of such huge models are too complex for systematic examination of every possible internal configuration to be possible (see Figure 1). 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.

If x is the probability that a randomly selected program input finds a fault then after N random inputs, the chances of the inputs not revealing a fault is $(1-x)^N$. Hence, the chances 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 (i.e. $x = 0.001$), moving C from 90% to 94% to 98% requires 2301, 2812, and 3910 black box probes (respectively) [2].
The problem of requiring exponential resources to evaluate software is not solved by using more informed evaluation method. For example, the infamous state space explosion problem imposes strict limits on how much a system can be explored via, say, automatic formal methods [3]. Also, with Bojan Cukic, we have explored numerous other examples where assessment effectiveness is exponential on effort [4].

Figure 1: Exponential cost of traditional assessment.

So, why does software work? One response might be to deny the premise of the question 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: for example, see the depressing litany of mistakes documented in Peter Neumann's "Risk Digest" [5]. However, given the internal complexities of software, it is puzzling that software doesn't crash more often.

Our thesis is that software works because internally it is surprisingly simple. This must be true since, if it were otherwise, then dumb apes like the authors (and, perhaps, the reader) would never have been able to build something as big as, say, the Internet or the software that controls the international passenger airline network.

*Submitted to IEEE Computer, October 2, 2006. An earlier draft is available on-line at `http://menzies.us/pdf/06strange.pdf`

†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.

Recent research results from artificial intelligence (AI) offer much support for our thesis that "software is simple." AI has discovered certain previously unrecognized regularities that can be used to quickly find solutions. For problems with those regularities, much of what can be found via complex and costly methods can also be found by random search. This is an important result since an incomplete randomized algorithm may also be the simplest algorithm available, or the fastest, or both [6]. This article presents examples of such algorithms, after some backgrounds notes on AI and recent empirical results in software analysis.

2 Background

Historically, AI has a bad reputation. It may therefore seem strange to assert that AI can help software analysis. To explain why we make this assertion, we begin with a careful review of some interesting results from that field.

A repeated conclusion in AI is that the behavior of large software is determined by a very small number of key variables, which we call the *collar* [7] variables. When collars are present, the problem of controlling software reduces to just the problem of controlling the variables in the collar.

Collars have been discovered and rediscovered in AI many times, and given different names including *variable subset selection* [8], *narrows* [9], *master-variables* [10], and *back doors* [11].

2.1 Variable Subset Selection

Numerous researchers have examined what happens when a data miner deliberately ignores some of the variables in the training data. For example, Ron Kohavi and George John studied a specific *variable subset selection* method. Their experiments show that, on 8 real world datasets, an average 81% 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 of 2.14% [8] (see Figure 2.1).

	av. number of variables			
dataset	before	after	$\frac{after}{before}$%	accuracy change
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	6%	+3.63%
horse-colic	22	2.8	13%	+1.63%
Pima	8	1	13%	+0.79%
sick-euthyroid	25	4	16%	+0.38%
soybean	35	12.7	36%	+0.15%
average	38.5	4.99	19%	+2.14%

Figure 2: Variable subset selection results; from [8]

2.2 Narrows

Amarel observed that search problems contain tiny collars (which he called *narrows*) in their search space which must be traversed in any solution. In such a search space, what matters is not so much how you get to these collars, but what decision you make when you get there. Since the route between collars is not important, Amarel's work defined macros encoding paths between them in the search space, effectively permitting a search engine to jump between them [9].

Note that Amarel's narrows would explain the variable subset selection results: variables from outside the narrows can be ignored without losing control of a system.

2.3 Master variables

James Crawford and Andrew Baker observed collars (which they called "master variables") while investigating different scheduling methods. They built a very fast complete search engine called TABLEAU and compared its performance to a very simple randomized search engine called ISAMP (shown in Figure 3). Both algorithms assign a value to one variable, then infer some consequences with forward checking. If contradictions are detected, TABLEAU backtracks while ISAMP simply starts over and re-assigns other variables randomly (giving up after `MAX-TRIES` number of times). Otherwise, they continue looping until all variables are assigned. Surprisingly, as shown in Figure 3, ISAMP took *less* time than

```
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	Time (sec)	% Success	Time (sec)	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 3: Average performance of TABLEAU vs ISAMP on 6 scheduling problems (A..F) with different levels of constraints and bottlenecks. From [10]. The `unit_ propagation` procedure of ISAMP is a special linear-time case of resolution; i.e. $(x) \wedge (\neg x \text{ } or \text{ } y_1 \text{ } ... \text{ } y_n) \vdash (y_1 \text{ } or ... \text{ } y_n)$ and $(\neg x) \wedge (x \text{ } or \text{ } y_1 \text{ } or ... \text{ } y_n) \vdash (y_1 \text{ } or ... \text{ } y_n)$.

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 the 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 since it restarts on every contradiction.

2.4 Back doors

Crawford and Baker argue that the collars/master variables play an important role in controlling how long it takes to find a solution. A similar conclusions 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. Constraining the collars also constrains the rest of the program (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 very quickly. They argue that this policy can reduce exponential time problems to polynomial time- provided that the collars can be cheaply located [11] (an issue we will return to below).

3 Collars (and Clumps)

Software is simple where it contains structures that dramatically reduce its internal structure. We have shown above that collars (by many names) have been observed many times in AI and that collars reduce the effective complexity of software. We will discuss below how to exploit collars for software analysis. Before doing so, however, we will look beyond the AI literature for evidence for collars, and collar-like, phenomena.

If the overall behavior of a software is determined by a small number of collar variables then we would expect three effects:

- Testing should quickly *saturate*; i.e. most program paths will get exercised early with little further improvement seen as testing continues.
- Random *mutation* is unlikely to find those variables and the net effect of those mutations would be very small (a *mutant* of a program is a syntactically valid, but randomly selected, variation to a program; e.g. swapping all plus signs to a minus sign). Hence, most mutations of software containing collars would result in the same effects.
- Software states should *clump*; i.e. only a small number of states will be reached at runtime. Collars imply clumps since the number of reachable states in a software will be quite small; i.e. just the number of possible settings to the collar (and not, e.g., the 2^{300} states mentioned in the introduction).

(Actually, clumps also imply collars since the collars are formed from the difference in the states reached at runtime. If the number of states is small then the number of differences will also be small.)

All these effects can be found in the literature. The saturation effect has been reported by Joseph Horgan and Aditya Mathur [12]. As to mutation testing, Christopher Michael found that in 80 to 90% of cases, there were no changes in the behavior of a range of programs despite numerous perturbations on data values using a program mutator [13]. In similar results, Eric Wong compared results using X% of a library of mutators, randomly selected ($X \in \{10,15,\ldots 40,100\}$). Most of what could be learned from the program could be learned using only X=10% of the mutators; i.e. after a very small number of mutators, new mutators acted in the same manner as previously used mutators [14]. The same observation has been made elsewhere by Timothy Budd [15] and Allen Acree [16].

As to clumping, Marek Druzdzel 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. Druzdel could show mathematically that there is nothing unusual about his application: we should always expect that software will clump (see Figure 4).

Empirical evidence for clumping comes from Radek Pelanek [17]'s detailed review of the structures of dozens of formal models. 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, the v also connects to u) and small "diameters" (i.e. 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; i.e. it would clump.

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

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

By taking logs of both sides, this equation becomes

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

The asymptotic behavior of such a sum of random variables is addressed by the central limit theorem. In the case where we know very 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, the expected case is that the frequency with which we reach states will exhibit a log-normal distribution; i.e. a small fraction of states can be expected to cover a large portion of the total probability space; and the remaining states have practically negligible probability.

Figure 4: Expected distribution of reachable states in software.

4 Software Analysis via Random Search

The key to exploiting collars and clumps is to first note that they dramatically reduce the search space within a program. With Singh, we have showed that the number of variables in a collar is expected to be very small (see Figure 5). The number of reachable states is set by the collar variables, so small collars also means that the number of clumping states will also be small. Hence, theoretically anyway, 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 [19]. Currently, we are experimenting with two random search algorithms: LURCH and TAR3. The results to date are quite promising. Some of those results are presented below.

4.1 TAR3

TAR3 is a randomized version of the TAR2 data miner [20] that 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.

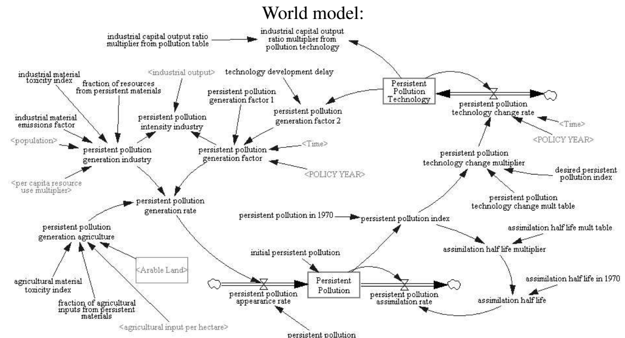
Consider the space of possible chains of inferences within software. Some of these chains intersect and may clash over the value of a variable at the intersection. We say that the collars contain the clashes that were not dependent on any other clashes. Let some goal in software be reachable by a narrow collar M or a wide collar N: i.e.

$$\left.\begin{array}{r} \xrightarrow{a_1} M_1 \\ \xrightarrow{a_2} M_2 \\ \dots \\ \xrightarrow{a_m} M_m \end{array}\right\} \xrightarrow{c} goal_i \xleftarrow{d} \left\{\begin{array}{l} N_1 \xleftarrow{b_1} \\ N_2 \xleftarrow{b_2} \\ N_3 \xleftarrow{b_2} \\ N_4 \xleftarrow{b_2} \\ \quad\quad \dots \\ N_n \xleftarrow{b_n} \end{array}\right.$$

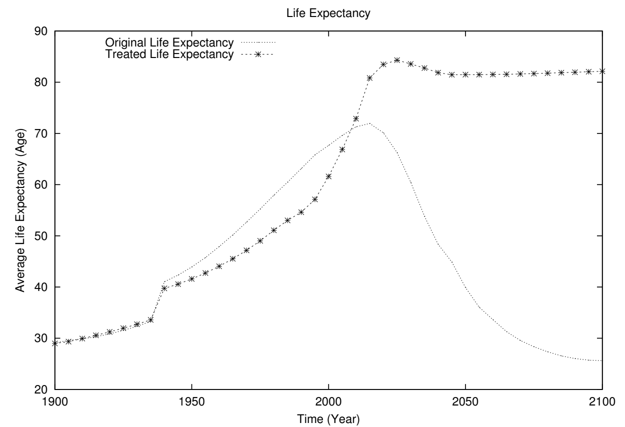
Let the cardinality of the narrow funnel and wide funnels be m and n respectively. Each m members of M is reached via a path with probability a_i while each n members of N is reached via a path with probability b_i. Two paths exist from the funnels to this goal: one from the narrow neck with probability 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 $narrow \gg wide$? In the case of uniform distributions of a_i, b_i where $\sum_{i=1}^{m} a_i = 1$, $\sum_{i=1}^{n} b_i = 1$, $a_i = \frac{1}{m}$, $b_i = \frac{1}{b}$ then we showed that at (e.g.) $m = 3$, the wider collar pathway is very unlikely. Precisely, the wider collar pathway is favored when $\frac{d}{c} \geq 1728$. i.e. only in the unlikely case that the d pathway is thousands of times more likely that c.
We built a small simulator to study the non-uniform case. The same conclusions were reached: narrow collars were millions of times more likely (for details, see [18]).

Figure 5: Mathematically, small clumps are likely.

To find the collar variables, TAR3 assumes that if collars exist, then they control the behavior of software. So, a random selection of the software behaviors must sample the collars (by definition). That is, we don't need to search for the collars - they'll find us. If we generate scenarios at random (e.g. via some Monte Carlo simulation), then score each run as "good" or "bad" using some domain knowledge (e.g. 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 at-



World population 1900 to 2100 with and without the learned treatment:



Learned decision tree (200 tests):

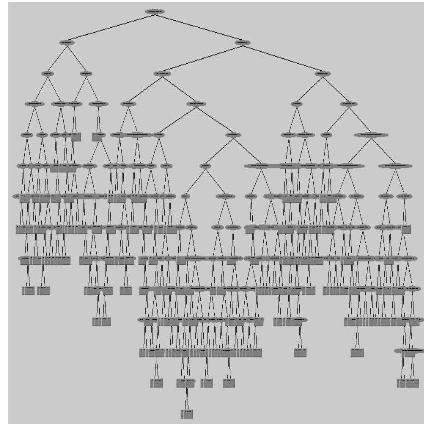


Figure 6: Output from TAR3, using data generated from the Limits to Growth Model.

tribute ranges that occur more in high scoring examples than in lower scoring examples.

Dustin Geletko [21] applied TAR3 to a version of the Limits to Growth [22] model shown at the top of Figure 6. This model studies the effects of the world's exponentially growing population and economy. The full model contains 295 variables and over 100 nodes (for space reasons, only a small portion of that model is shown in Figure 6). MIT faculty studied this model for several years to find factors that prevented global population over-shoot and collapse; i.e. *desired completed family size normal* = 0..2 and *industrial capital output ratio* = 3..5. The two effects seen in the Limits to Growth Study were also seen in the UCI data. TAR3 found the same factors in a 30 minute run. Most of that time was spent generating the example data from the model; the learning only took seconds. To check TAR3's conclusions, another simulation was conducted where the inputs were constrained according to TAR3's recommendations. The results are shown as the middle plot of Figure 6: average life expectancy in the year 2100 increases from 30 to 80 years and appears to be stable from that time onwards.

Since TAR3 returns the collar variables and since we expect the number of collar variables 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 used by TAR3. That learner returned a decision tree with 200 tests, shown bottom of Figure 6. In a result consistent with the collar hypothesis, TAR3's theory learned from the same data only needed to test two variables: *desired completed family size normal* and *industrial capital output ratio*.

The two main effects of this study were that (1) TAR3 returned very small theories and (2) those theories were effective in changing the distribution of some system. Ying Hu [23] describes numerous studies with the algorithm and multiple datasets from the standard UCI data mining data sets [24] (plus some software engineering domains). The two effects seen in the Limits to Growth Study were also seen 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.

4.2 LURCH

LURCH is a design debugging tool for state transition diagrams [27, 25]. If more than one transition can be applied at any time, LURCH selects one at random. The program never backtracks; rather it generates state transitions as fast as possible. After some termination condition (e.g. looping, too many transitions), LURCH resets and generates another randomly selected trajectory of states.

For example, Figure 7 compares the runtimes and memory required for a complete search and randomized search to solve the "N-queens" problem (place N queens on an $N * N$ chess board such that no queen can take any other). For the complete search, the SPIN model checker [26] was run in six different modes that tried various methods to improve that search. For the random search, LURCH was used to randomly explore a finite state machine model of N-queens.

Crawford and Baker commented above that a complete search looks into everything and can get stuck in some irrelevant corner of the problem. This effect can be seen in Figure 7: SPIN's complete search gave up and died on anything larger than 15*15 (shown by the vertical dashed line). But just like ISAMP, LURCH's random search has a built-in "get out jail free" card: when it gets stuck, it can (metaphorically) jump over a wall and start afresh somewhere else. Observe how LURCH's search scaled to much larger problems than SPIN.

Figure 7 also demonstrates the *order effects* that plague deterministic search. For deterministic algorithms, certain inputs always result in slowest runtimes. For example, insertion sort runs slowest if the inputs are already sorted in reverse order. For another example, in Figure 7, SPIN's complete complete search takes exponentially less time and memory for boards of odd size (e.g. $N = 11, 13, 15$) than boards of even size (e.g. $N = 10, 12, 14$). The random search, on the other hand, jumps around the input data, so it can be difficult to find inputs that generates worst-case runtimes.

A standard objection to using random search (and systems like LURCH) is that they can miss some safety critical property violations. In this regard, our recent work with Duran Desivksi and Bojan Cukic is insightful [25]. In that study, a formal requirements model was explored with SPIN and LURCH. Both tools missed property violations that the other found. LURCH missed violations

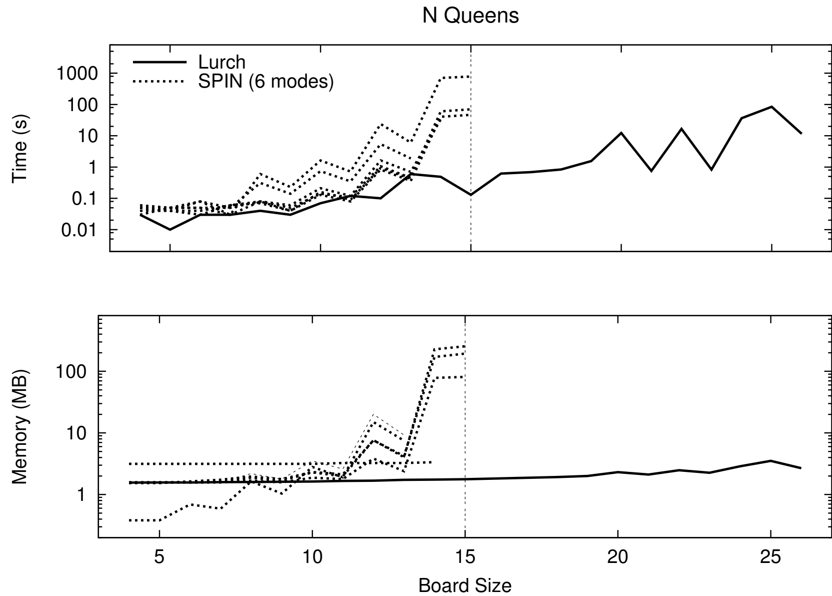


Figure 7: N-queens using complete (SPIN) vs random (LURCH) search.

because of its random search and SPIN missed properties because of a heuristic used at runtime to let it scale up to that model. Hence, we recommend *combining* random with complete search methods.

5 Discussion

The strangest thing about software is that it works at all. Yet it does work, so *something* must be simplifying the internals of our software.

Here we have argued from AI, and some other results, that much of the internal structure of software is constrained and simplified by collars and clumps. With that as a premise, we have been exploring random search tools for software analysis. Two such tools are the TAR3 contrast set learner and the LURCH design debugger. For software with clumps and collars, these tools reveal much of what can be revealed using more complete methods.

To be sure, there are times when random search is dangerous and should be avoided. For example, the software controller of the ascent stage of a manned spacecraft 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 on-board diagnosis when the craft is (a) in deep space and (b) in deep trouble and (c) it takes too long to ask for help from ground control.

In other words, complete methods and random methods should be mixed and matched. For the mission-critical kernel of the software, tools like SPIN should be used for complete validation. But the rest of the software may be too big for complete analysis, in which case a random search (e.g. using LURCH and/or TAR3) may be the only cost-effective option.

We would recommend that software analysis starts with random analysis first (since it is so cheap) and only move to the more complex methods when random methods run out of steam. Writing nearly two decades ago, Barry Boehm made a analogous proposal for iterative software exploration in his famous paper “A Spiral Model for Software Development” [28]. Writing at the same time, Donald Norman argued in “The Design of Everyday Things” [29] that such iterative exploration is essential in

any human design process.

Since 1988, much has changed. 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 the different clumps. This kind of randomized search and learning shows great promise for finding the key decisions within seemingly intricate software.

References

[1] T.J. Menzies, 'Principles for generalised testing of knowledge bases', *Ph.D. Thesis*, University of New South Wales, 1995. Ph.D. thesis. Available from http://menzies.us/pdf/95thesis.pdf.

[2] J.M. Voas and K.W. Miller, 'Software testability: The new verification', *IEEE Software*, 17–28 (1995). Available from http://www.cigital.com/papers/download/ieeesoftware95.ps.

[3] M. Lowry, M. Boyd, and D. Kulkarni, 'Towards a theory for integration of mathematical verification and empirical testing', *Proceedings, ASE'98: Automated Software Engineering*, 1998, pp. 322–331.

[4] T.J. Menzies and B. Cukic, 'How many tests are enough?', S.K. Chang (ed.), *Handbook of Software Engineering and Knowledge Engineering, Volume II*, 2002. Available from http://menzies.us/pdf/00ntests.pdf.

[5] Peter G. Neumann, *Computer-Related Risks*, ACM Press / Addison Wesley, 1995.

[6] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995. (reprinted 1997,2000).

[7] T. Menzies and J. Richardson, 'Making sense of requirements, sooner', *IEEE Computer* (2006). Available from http://menzies.us/pdf/06qrre.pdf.

[8] Ron Kohavi and George H. John, 'Wrappers for feature subset selection', *Artificial Intelligence*, **97**(1-2), 273–324 (1997).

[9] S. Amarel, 'Program synthesis as a theory formation task: Problem representations and solution methods', in R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach: Volume II*, Kaufmann, Los Altos, CA, 1986, pp. 499–569.

[10] J. Crawford and A. Baker, 'Experimental results on the application of satisfiability algorithms to scheduling problems', *AAAI '94*, 1994.

[11] R. Williams, C.P. Gomes, and B. Selman, 'Backdoors to typical case complexity', *Proceedings of IJCAI 2003*, 2003. http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf.

[12] J. Horgan and A. Mathur, 'Software testing and reliability', M. R. Lyu (ed.), *The Handbook of Software Reliability Engineering*, McGraw-Hill, 1996, pp. 531–565.

[13] C.C. Michael, 'On the uniformity of error propagation in software', *Proceedings of the 12th Annual Confererence on Computer Assurance (COMPASS '97) Gaithersburg, MD*, 1997.

[14] W.E. Wong and A.P. Mathur, 'Reducing the cost of mutation testing: An empirical study', *The Journal of Systems and Software*, **31**(3), 185–196 (1995).

[15] T.A. Budd, 'Mutation analysis of programs test data', *Ph.D. Thesis*, Yale University, 1980.

[16] A.T. Acree, 'On mutations', *Ph.D. Thesis*, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[17] R. Pelanek, 'Typical structural properties of state spaces', *Proceedings SPIN'04 Workshop*, 2004. Available from http://www.fi.muni.cz/~xpelanek/publications/state_spaces.ps.

[18] T. Menzies and H. Singh, 'Many maybes mean (mostly) the same thing', M. Madravio (ed.), *Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from http://menzies.us/pdf/03maybe.pdf.

[19] T. Menzies and C.C. Michael, 'Fewer slices of pie: Optimising mutation testing via abduction', *SEKE '99, June 17-19, Kaiserslautern, Germany. Available from `http://menzies.us/pdf/99seke.pdf`*, 1999.

[20] T. Menzies, J. Di Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and Davis J, 'When can we test less?', *IEEE Metrics'03*, 2003. Available from `http://menzies.us/pdf/03metrics.pdf`.

[21] D. Geletko and T. Menzies, 'Model-based software testing via treatment learning', *IEEE NASE SEW 2003*, 2003. Available from `http://menzies.us/pdf/03radar.pdf`.

[22] D.H. Meadows, D.L. Meadows, J. Randers, and W.W. Behrens, *The Limits to Growth*, Potomac Associates, 1972.

[23] Y. Hu, 'Treatment learning: Implementation and application', *Master's Thesis*, Department of Electrical Engineering, University of British Columbia, 2003. Masters Thesis.

[24] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. URL: `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[25] D. Desovski D. Owen and B. Cukic, 'Effectively combining software verification strategies: Understanding different assumptions', *ISSRE 06*, 2006.

[26] G.J. Holzmann, 'The model checker SPIN', *IEEE Transactions on Software Engineering*, **23**(5), 279–295 (1997).

[27] David Owen, Tim Menzies, Mats Heimdahl, and Jimin Gao, 'On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate', *IEEE NASE SEW 2003*, 2003. Available from `http://menzies.us/pdf/03lurchc.pdf`.

[28] B. Boehm, 'A spiral model of software development and enhancement', *Software Engineering Notes*, **11**(4), 22 (1986).

[29] D.A. Norman, *The Design of Everyday Things*, DoubleDay Currency, 1989.