Maintaining Maintainability = Recognizing Reachability

Tim Menzies
NASA/WVU IV&V Facility
100 University Drive, Fairmont WV 26554, USA
tim@menzies.com;http://www.tim.menzies.com

Bojan Cukic
Dept. Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV USA
cukic@csee.wvu.edu

1 INTRODUCTION

How do we assess the maintainablilty of our systems? Given the infinity of parameters we could extract from our programs, what is the least we must collect to tell us the most about maintainablilty? To answer this question, we offer a theoretical model of maintenance effort. We will assume that:

- One of the main cost of maintenance is continual retesting. Testing will be characterized as the construction of pathways that *reach* from inputs to some interesting zone of a program. This zone could be a bug or a desired feature. In this *reachability* view, the goal of testing is to show that a test set uncovers no bugs while reaching all desired features.
- A system is hard to maintain if it is hard to test; i.e. if the reachability odds are low. In this view, easy maintenance is impossible without easy testing and easy testing is impossible without easy reachability.

After building and simulating a model of system reachability, we will use a sensitivity analysis to find the *grasp* of a program; i.e. the key parameters that are the main drivers that change system reachability. In this article, we show that this reachability distribution has steep cliffs and wide low plains. Choices made by the system designer can change a program's *grasp* and drive the system over the cliffs. Other choices can keep the system on the low plains where the probability of detecting faults using cheap black-box probes is very high.

The rest of this article is structured as follows. §2 sets out some general background to this work. §3 describes our reachability model. §4 shows the output from that model and uncovers the parameters that define a program's *grasp*. §5 uses a literature review to argue that our reachability model is producing sensible results. §6 discusses the use of the model to avoid the reachability cliffs.

Draft of a paper submitted to the International Workshop on Empirical Studies of Software Maintenance (WESS 2000) http://hometown.aol.com/GEShome/wesscfp.htm *October 14, 2000 San Jose, CA*

2 PREAMBLE

We begin by noting two limitations to this analysis. This paper presents an average-case analysis of reachability. By definition, such an average case analysis says little about extreme cases of high criticality. Hence, this approach is only applicable in situations where cost-cutting in software assessment is justified. That is, our analysis is not applicable for safety-critical systems. Also, this analysis only detects when maintenance is hard. It is silent on when maintenance is easy. For example, a system with easy reachability may still be hard to maintain if, say, the programming team keeps changing.

Our analysis of cost-effective testing assumes random test inputs. The rest of this section explains why we make that assumption. We start with Equation 1 that compares the start-up costs required for different testing procedures [15].

$$Start\ up\ costs \quad : \quad BB < WB \ll FM \tag{1}$$

Black-box (BB) methods are the cheapest: input sets can be quickly built using automatic random selection of data (possibly from an operational profile). White-box (WB) testing costs more than black-box to define: analysts must reflect over the internals of a program to invent test inputs that exercise K different partitions within a program. Formal methods (FM) are by far the most expensive since, after the program or specification is understood, we must write its representation. This formal representation contains the essential features of the specification. It is a formal model; i.e. all its constructs have a precise semantics which can be revealed by automatic methods. Building formal representation is time-consuming and, in the case of automatic formal methods like SPIN [12], requires scarce and specialized Ph.D.-level skills.

Two methods that avoid this pessimism are automatic code generation (ACG) [5] and software fault trees (SFTs) [14]. The dream of ACG is that systems will never need testing since there were generated from perfect components using some perfect automatic method. Research in this area is exciting but not widely applicable since the technology currently available for automatic generation is not perfect. Further, experience strongly suggests that our COTS components are far from perfect. SFTs assume a very simple language for the formal model which is applied to only parts of

a system. Levenson's case studies [14] suggest that a small number of heuristically guided probes into a program using some rapidly generated search criteria (the SFTs) finds as many errors in less time than traditional rigorous formal methods. Note that this result endorses heuristic approaches to testing, which is the theme of this paper.

$$Detecting\ errors \quad : \quad \frac{WB}{K} \leq BB \leq WB < FM \quad (2)$$

Equation 2 says that the more expensive test methods can detect more errors. Formal methods can find the most errors since a single formal first-order query is equivalent to many white-box or black-box test inputs. Also, white-box *partitioning* can find errors faster than black-box probes. Partitioning divides the input space into K partitions. Each partition K_i exercises one desired feature of a program. Once the partitions are created, ideally, we need only run one test in each partition. The bad news is that creating such partitions is a non-trivial task and adds much to the white-box costs shown in Equation 1. Also, when analyzing system reliability, it is not good practice to only analyze the situations defined by the analysts [10, p670]. Randomized selection of test inputs may uncover errors that could be missed if testing is biased by the incorrect assumptions of the analysts [13]. Further, a repeated mathematical result is that the odds of detecting an error with white-box probing is nearly the same as with black-box probing [11]. Even assuming certain special cases that favor white-box probes (e.g. all inputs are equally likely), white-box using K partitions is only ever K times better than black-box at finding errors [10].

$$Test\ item\ size \quad : \quad BB \approx WB \gg FM \quad (3)$$

Equation 3 is another argument for favoring black-box methods. As detection power increases (Equation 2), the computational cost also increases. Except for trivially simple programs, no test procedure can ever be exhaustive. Tests exercise pathways in programs. Gabow et.al. [9] showed that building pathways across programs with impossible pairs (e.g. some boolean and its negation) is NP-complete for all but the simplest programs[1]. NP-complete tasks must be solved using incomplete heuristics since the runtimes and memory requirements of a complete solution can explode exponentially. As evidence of this, in one case study at NASA, it was found that invariants from 30 JAVA classes take one gigabyte of RAM to check with automatic formal methods. Consequently, as we increase thoroughness of testing, we must restrict the size of the item being tested. One of the "black arts" in formal methods is *abstraction*; i.e. the summarization of the essential features model in a form succinct enough to be thoroughly searched by automatic formal methods like SPIN. In practice, abstraction may take months of work, must be repeated if the system ever changes significantly, and so can contribute significantly to the costs of Equation 1.

[1] A program is very simple if it is very small, or it is a simple tree, or it has a dependency networks with out-degree ≤ 1.

$$Localizing\ errors \quad : \quad BB < WB < FM \quad (4)$$

Equation 4 is the main reason for avoiding black-box testing, despite Equation 1 and Equation 3. Once an analyst detects the error, she/he must trace backwards through the system to localize the source of the error. By definition, a black-box test gives no insight into the internal structure of a system. Hence, the only way to localize bugs with black-box methods is to recursively disable half the program (picked at random) and see if the bug still persists in the remaining portion. In the case where each black-box test is unlikely to find an error, this may be a long and tedious task. In contrast, the other methods give extensive support for localization. The search for localizing bugs found using partition K_i can be constrained only to those parts of a program exercised by K_i. Also some automatic formal methods generate counter-examples; i.e. exact traces showing how some constraint was violated. Given such a counter-example, fault localization can be constrained to the program zones found in the trace.

In summary, a choice of testing strategies requires trade-offs between the size of the item being tested, the resources available for testing, the power of a test to find errors, and the support offered for fault localization. Consider the hard case of maintaining rapidly changing systems built from COTS packages. The time-consuming construction of formal models may be impractical. Further, given the black-box nature of COTS packages, we may not know enough about the system to build its formal model. Hence, in this hard case, we must choose between black-box and white-box methods. We argue below that, in certain cases, the probability of black-box probes finding errors is high. In this case, the drawbacks noted in Equation 2 and Equation 4 would be removed; i.e. the extra detection power of formal methods would be not be required and black-box fault localization would not be slow or tedious. Without those drawbacks, then Equation 1 and Equation 3 make a compelling case for black-box testing.

3 REACHABILITY THEORY

This section shows how to calculate the number of tests required to reach any part of the program, given random inputs.

We represent programs as directed graphs indicating dependencies between variables in a global symbol table; see Figure 1.B. Such graphs have V nodes which are divided into $andf\%$ and-nodes and $(orf = 1 - andf)\%$ or-nodes. These nodes are connected by yes-edges, which denote valid inferences, and no-edges, which denote invalid inferences. For the sake of clarity, we only show the yes-edges in Figure 1.B. To add the no-edges, we find every impossible pair and connect them; e.g. we link `A=1` to `A=2` with a no-edge since we cannot believe in two different assignments for the same variable at the same time. We say that, on average, each node is touched by no_μ no-edges.

```
byte a=1; byte b=1; bit f=1;
active proctype A(){
 do :: f==1 -> if ::a==1 -> a=2;
                  ::a==2 -> a=3;
                  ::a==3 -> f=0; a=1;
               fi  od}
active proctype B(){
 do :: f==0 -> if ::b==1 -> b=2;
                  ::b==2 -> b=3;
                  ::b==3 -> f=1; b=1;
               fi  od}
```

Figure 1.A: *Two procedural subroutines from the SPIN model checker [12].*

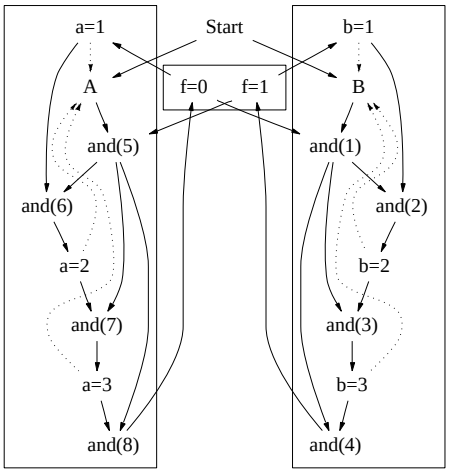

Figure 1.B: *And-or links found in Figure 1.A.* `Proctype A` *and* `B` *are shown in the boxes on the left-hand side and right-hand side respectively. These* `proctypes` *communicate via the shared variable* `f`*. Dotted lines represent indeterminism; e.g. after* `proctype A` *sets* `a=2`*, then we can go to any other statement in the proctype.*

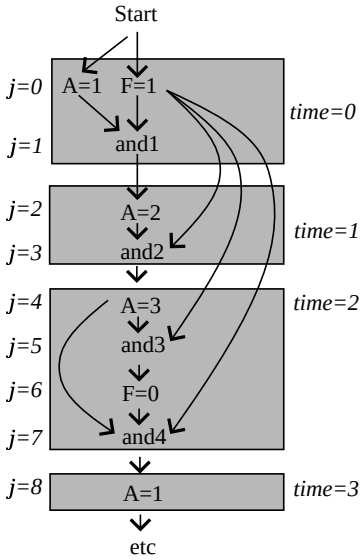

Figure 1.C: *Example reachability network built from Figure 1.B showing how to violate property* `always not A=3` *at height* $j = 4$.

Figure 1: From code samples (top) to and-or graph (middle) to reachability network (bottom).

When inferencing across these graphs, variables can be assigned, at most T different values; i.e. a different assignment for each time tick T in the execution (classic propositional systems assume $T = 1$; simulation systems assume $T \geq 1$). Nodes are reached across the dependency graph using a network of height j where the inputs are at height $j = 0$. For example, Figure 1.C shows a the network of height $j = 8$ that reaches $A = 1$ after reaching $A = 1$, then $A = 2$, then $A = 3$. Note that since four values are assigned to A, then time T must be 4.

In the reachability model, we say that the and-nodes and or-nodes have the mean munber of parents $andp$ and orp, respectively. An or-node contradicts, on average, no other or-nodes. $andp, orp, no$ are random gamma variables with means $andf_\mu, andp_\mu, orp_\mu, no_\mu$; "skews" $andp_\alpha, orp_\alpha, no_\alpha$; and range $0 \leq \gamma \leq \infty$. $andf$ is a random beta variable with mean $andf_\mu$ and range $0 \leq \beta \leq 1$. And-nodes are reached at height j via one parent at height $i = j - 1$ and all others at height

$$i = \beta(depth) * (j - 1), \tag{5}$$

so $0 \leq i \leq (j - 1)$. Note that as $depth$ decreases, and-nodes find their pre-conditions closer and closer to the inputs. The probability $P[j]_{and}$ of reaching an and-node at height $j > 0$ is the probability that one of its parents is reached at height $j - 1$ and the rest are reached at height $1..(j - 1)$; i.e.

$$P[j]_{and} = P[j-1] * \left(\prod_{2}^{andp[j]} P[i] \right) \tag{6}$$

Or-nodes are reached at height j via one parent at height $i = j - 1$. The probability $P[j]_{or}$ of reaching an or-node at height $j > 0$ is the probability of not missing any of its parents; i.e.

$$P[j]_{or} = 1 - (1 - P[j-1]) * \left(\prod_{2}^{orp[j]} (1 - P[i]) \right) \tag{7}$$

The probability $P[j]$ of reaching any node is hence the sum of $P[j]_{or}$ and $P[j]_{and}$ weighted by the frequencies of and-nodes and or-nodes; i.e.

$$P[j] = andf[j] * P[j]_{and} + orf[j] * P[j]_{or} \tag{8}$$

To compute $P[j]$ for multiples goals, we add an small sub-graph represented the conjunctive normal form for that combination of goals. This sub-graph will add two more layers to the existing program graph; i.e. after proving a single goal at height j, we can prove multiple goals at height $j + 2$.

Other details not shown above are loop and contradiction detection. See [17] for those details.

4 EXPERIMENTS WITH REACHABILITY

A simulation of the above system of equations requires around 200 lines of Prolog. We can execute the simulation

Figure 2.A:
400 runs, in=1..50, j=1..10

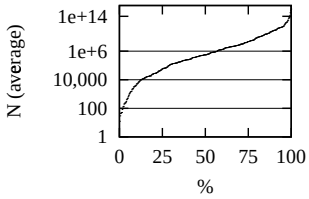

Figure 2.B:
400 runs, in=950..1000, j=1..10

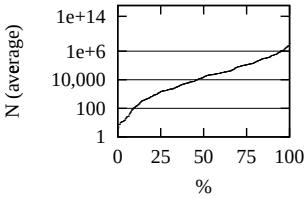

Figure 2.C:
400 runs, in=450..550, j=40..50

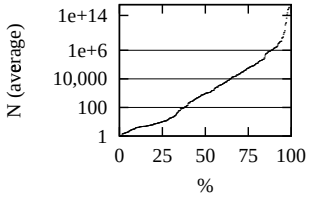

Figure 2.D:
400 runs, in=450..550,j=90..100

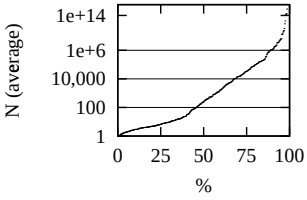

Figure 2: Outputs from the reachability model, restricted to certain ranges. Y-axis comes from Equation 10; X-axis generated by sorting simulation outputs.

and report the $P[j]$ figures as the number of tests N required to be 99% percent certain of reaching a random node in a dependency graph using randomly selected inputs. N randomly selected inputs have certainty

$$C = 0.99 = 1 - \left((1 - P[j])^N\right) \quad (9)$$

of reaching a node in dependency graph. Hence,

$$N = \frac{log(1 - 0.99)}{log(1 - P[j])} \quad (10)$$

The above model was simulated for a wide range of the parameters; e.g., up to 10^8 nodes, up to 1000 inputs, up to 100 time ticks, wildly varying the frequency and skew of and-nodes, or-nodes, and no-edges, etc. The frequency distribution of the calculated N values for various values of hight j and the number of inputs in are shown in Figure 2. Note that there are many cases where much of a program is reachable. For example, as j increases, more and more of the system is reachable. By $j = 90..100$ (see Figure 2.D), nearly 50% of the the program is reachable with less than 100 randomly selected tests; nearly 75% is reachable with less than 10,000 randomly selected inputs; and only 10% of the program is out of reach of up to 1,000,000 randomly generated inputs. Note that these random tests can be performed in parallel. Given that the standard desktop machine is now a 500MHz box, then we are confident that in many situations, organizations can exercise enough random inputs to achieve up to 90% reachability.

A machine learner (C4.5 [21]) was used to perform a sensitivity analysis. The simulation outputs were classified using Equation 10 as follows:

		Subsets		
Variable	Description	**All**	**Some**	**Least**
$time$	time ticks	√	√	√
j	height	√	√	√
$depth$	mean relative parent height (β)	√	√	√
orp_μ	$\gamma(orp)$ mean	√	√	√
$andp_\mu$	$\gamma(andp)$ mean	√	√	√
$andf_\mu$	mean and node frequency (β)	√	√	
$andp_\alpha$	$\gamma(andp)$ skew	√		
orp_α	$\gamma(orp)$ skew	√		
no_α	$\gamma(no)$ skew	√		
no_μ	$\gamma(no)$ mean	√		
in	number of inputs	√		
v	number of nodes	√		
$isTree?$	0,1	√		
classes	$1 \ldots 10^2$ or $10^2 \ldots 10^4$ or $10^4 \ldots 10^6$ or $10^6 \ldots \infty$	√	√	√

Figure 3: Three subsets of the model parameters used in the sensitivity analysis.

$$Class = \begin{cases} fast\,and\,cheap & if\ N < 10^2, \\ fast\,and\,moderately\,expensive & if\ N < 10^4, \\ slow\,and\,expensive & if\ N < 10^6, \\ impossible & otherwise. \end{cases}$$

Decision trees to predict these classifications were built using three different subsets of the model parameters (see Figure 3). For each subset, learners were given example sets of different sizes: 150 examples, 1500 examples, and 150000 examples. A baseline for classifier accuracy was generated by building a classifier using **All** 13 model parameters. A nearly similar classifier could be built by ignoring the number of inputs, the size of the program, the skews in the distributions, and information about the no-edges (evidence: compare the **All** curve to the **Some** curve in Figure 4)[2]. However, if we blocked access by the machine learner to $andf$, the the accuracy of the classifier feel by 15-20% (evidence: compare the **Some** curve to the **Least** curve in Figure 4). Hence, we say that the **Some** set represents the *grasp* of a program well; i.e. this is the minimal set of parameters needed to compute reachability in our model.

5 SUPPORTING EVIDENCE

The above theoretical results are consistent with numerous empirical testing results. Elsewhere, we have catalogued the number of tests used to certify expert systems. An often repeated observation is that a small number of inputs can often reach significant errors in a program [3, 16]. Levenson's work on software fault trees suggest the same conclusion for conventional software [14]. These results support the observation of $j \geq 90$ reachability trees in which much of the program is reachable using a not-large number of inputs.

[2] Space does not permit us to show the 30,000 decision tree nodes found in classsifier learnt from the **Some** set. A heavily pruned version of that **Some** tree is shown in Figure 5 (for a larger, and hence more accurate, classifier, see [17]).

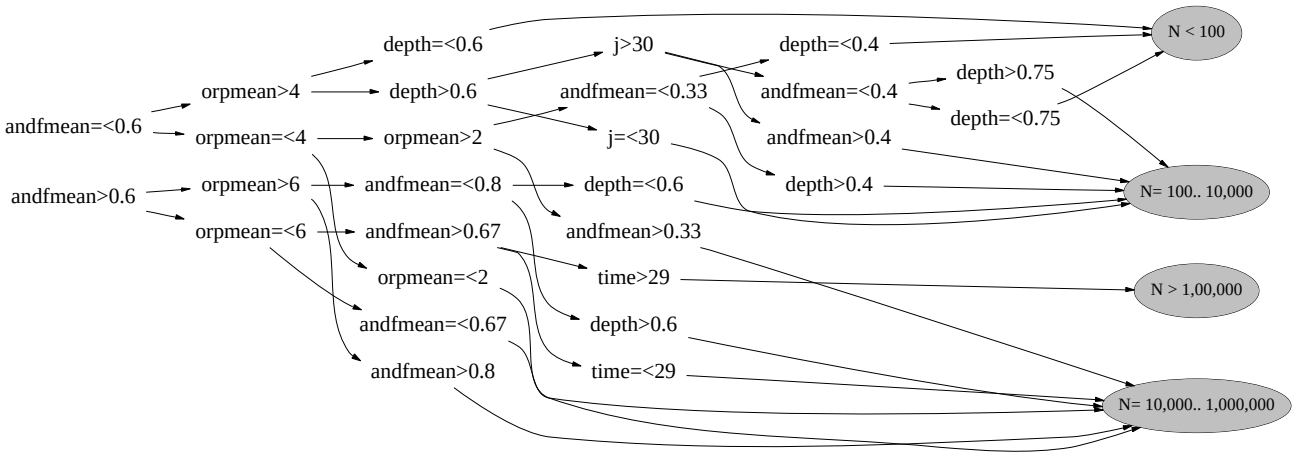

Figure 5: A classifier for predicting the number of random tests N required for 99% probability of full reachability (learnt from the **Some** set from Figure 3). For space reasons, this classifier is heavily pruned: see [17] for a more accurate classifier.

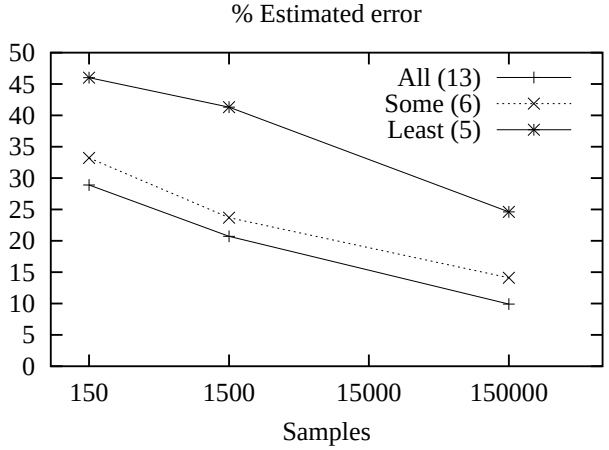

Figure 4: Results of the sensitivity analysis, built from the subsets shown in Figure 3.

Rothermel et.al. [22] report that reducing test suite size significantly decreased the number of faults detected. The same effect can be seen in our simulations outputs. As the input size is increased from Figure 2.A to Figure 2.B, the nodes reached by 10,000 tests increases from 10% to 50%. However, our results suggest that the conclusions of the Rothermel et.al. study can be generalized. They found that test suite size was one factor in assessing testability. We found numerous interactions between 13 factors of Figure 3, including test suite size, that influence testability.

Numerous researchers in mutation testing report that most mutations give rise to the same nominal and off-nominal behaviours [1,4,20,24]. This is consistent with mutators not effecting the *grasp* parameters (e.g. the and-node frequency).

Various researchers have noted that the portions of a program used in normal operation are a small subset of the total program [2, 6], and that test coverage suites often do not target the entire program [8, 13]. This is consistent with the hypothesis that programs include easily reachable and very unreachable zones. The same effect can be seen in our simulation outputs. Recall that the y-axis of Figure 2 is a logarithmic scale: as our curves rise on that scale, they are escalating into very unreachable zones.

Researchers in AI and requirements engineering explore inconsistent and indeterminate theories. A repeated result is that committing to a randomly selected resolution to a conflict reaches as much of a program as carefully exploring all resolutions to all conflicts [7, 18, 19, 23] . This is consistent with the sensitivity results shown in Figure 4. Recall that we could ignore the no parameters in the **Some** set and still build a nearly optimum classifier. That is, the added complexity of exploring a space of conflicting options is not a crucial factor in determining reachability.

6 CONCLUSION

Based on a theoretical model, we say that the reachability *grasp* (reachability) of a program can be computed from the six parameters in the **Same** set of Figure 3. These six parameters are the minimal set we need to detect hard reachability, hence hard testability, hence hard maintainability. Of those parameters, four can be detected via a static analysis while two ($depth$ and j) must be computed from a running program.

One application of this work is assessing the maintainabilty of COTS systems. Suppose each COSTS package had an API that returned that package's *grasp*. Analysts could then use the grasp to assess the reachability, hence testability, hence maintainability, of a COTS package without revealing the internal secrets of a vendor's system. With knowledge of the *grasp* parameters, we can detect if our program is veering over the cliffs of reachability. For example, Figure 5 tells us that rhere are three ways to show that less than 100 tests are required to reach most parts of a program. The frequency of the and-nodes appears in all three paths is:

$$(andf_\mu \leq 0.33) \wedge (andf_\mu \leq 0.4) \wedge (andf_\mu \leq 0.6) = andf_\mu \leq 0.6$$

That is, if $andf_\mu$ ever rises above 0.6, there is no way that 100 random tests will be enough to test that program. Figure 5 also tells us that there is only one way to show that more than 1,000,000 tests are required to reach most parts of a program. The and-node frequencies in that path are:

$$(andf_\mu > 0.6) \wedge (andf_\mu > 0.67) = andf_\mu > 0.67$$

That is, if $andf_\mu$ rises from 0.6 to 0.67, then a system with easy reachability could suddenly transform into a system with hard reachability, hence hard testability, hence hard maintainability.

ACKNOWLEDGEMENTS

This work was partially supported by NASA through cooperative agreement #NCC 2-979.

REFERENCES

[1] A. Acree. *On Mutations.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[2] A. Avritzer, J. Ros, and E. Weyuker. Reliability of rule-based systems. *IEEE Software*, pages 76–82, September 1996.

[3] J. Bieman and J. Schultz. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, 7(1):43–51, 1992.

[4] T. Budd. *Mutation analysis of programs test data.* PhD thesis, Yale University, 1980.

[5] W. Buntime. Will domain-specific code syhthsis become a silver bullet? *IEEE Intelligent Systems*, pages 9–15, March/April 1998.

[6] R. Colomb. Representation of propositional expert systems as partial functions. Artificial Intelligence (to appear), 1999. Available from `http://www.csee.uq.edu.au/~colomb/PartialFunctions.html`.

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

[8] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach.* International Thompson Press, 1997.

[9] H. Gabow, S. Maheshwari, and L. Osterweil. On two problems in the generation of program test paths. *IEEE Trans. Software Engrg*, SE-2:227–231, 1976.

[10] W. Gutjhar. Partition vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, September/October 1999.

[11] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

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

[13] J. Horgan and A. Mathur. Software testing and reliability. In M. R. Lyu, editor, *The Handbook of Software Reliability Engineering*, pages 531–565, McGraw-Hill, 1996.

[14] N. Levenson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.

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

[16] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. (to appear).

[17] T. Menzies, B. Cukic, H. Singh, and J. Powell. Testing indeterminate systems, 2000. ISSRE 2000 (to appear).

[18] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *RE '99*, 1999. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-009.pdf`.

[19] T. Menzies and C. Michael. Fewer slices of pie: Optimising mutation testing via abduction. In *SEKE '99, June 17-19, Kaiserslautern, Germany. Available from* `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-007.pdf`, 1999.

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

[21] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[22] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimizaton on the fault-detection capabilities of test suites. In *Proceedings of International Conference on Software Maintenance '98*, pages 34–43, November 1998. Available from `http://www.cis.ohio-state.edu/~harrold/research/webpapers/icsm98-min.pd%f`.

[23] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings, AAAI '96*, pages 971–978, 1996.

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