Submitted to the Handbook of Software Engineering and Knowledge Engineering,
Volume 2 (see http://www.ksi.edu/seke/hand.html)
URL= http://tim.menzies.com/pdf/00ntests.pdf.

How Many Tests are Enough?

Tim Menzies[‡], Bojan Cukic[¶]

[‡]Department of Electrical & Computer Engineering
University of British Columbia, Canada
[¶]Department of Computer Science and Electrical Engineering
West Virginia University, USA
<tim@menzies.com; cukic@csee.wvu.edu>

November 28, 2000

1. Introduction

How many tests will be required to test software? At first glance, we might think that an impossibly large number of tests are required. A system containing V variables with S assignments may require one test for each combination of assignments; i.e.

$$\#tests \ = \ N = \ S^V \tag{1}$$

It is simple to show that this is an impossibly large number of tests. Consider one sample of fielded expert systems in which knowledge bases were found to contain between 55 and 510 "literals" [49]. Literals offer two assignments for each proposition: true or false; i.e. $S = 2$ and V is half the number of literals. Assuming:

- It takes one minute to consider each test result (which is a gross underestimate), and
- The effective working year is 225 six hour days,

then a test of those sampled systems would take between 29 years and 10^{70} years (a time longer than the age of this universe).

The goal of this chapter is to offer an optimistic alternative to the pessimism of Equation 1. After a review of the mathematics of testing, this chapter will conclude that in many cases, effective testing may need far fewer tests than Equation 1. Various testing regimes will be discussed:

Black box methods (BB): In BB testing, input sets can be quickly built using automatic random selection of data. For nominal BB testing, these inputs are drawn from an operational profile describing the normal environment of a system [47]. For off-nominal BB testing, these inputs can come from within and without the operational profile. The discussion will focus on two special types of BB testing:

- If developers only require an approximate assessment of a system, then **approximate testing** can suffice.
- Depending on the goal of the testing, we can use **sampling theory** to explore less than S^V states of a program.
- Various **stopping rules** can be applied to control the time spent for testing.

White box methods (WB): In WB partition testing, analysts reflect over the internals of a program to invent test inputs that exercise K different *partitions* of a program. Each partition represents one class of behavior of the system.

Formal methods (FM): In FM testing, after the program or specification is understood, analysts 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. Let x denote the level of rigor and $F^x M$ denote a formal method with level-of-rigor x. At least four different styles of FM can be found in the literature:

- $F^{-1}M$ refers to very lightweight formal methods.
- $F^0 M$ refers to manual formal methods that rely heavily on mathematical representations.
- $F^1 M$ are the most common type of formal methods in use today. In $F^1 M$ testing, programs are written using conventional methods, then automatic formal methods are used to debug and revise the program.
- $F^2 M$ refers to full life cycle formal methods in which code is generated automatically from large libraries of formally proved components.

Before beginning, it is important to note that there are many methods of testing software other than those listed above (e.g. see the excellent discussions in [21, 35])*. This chapter focuses on the methods listed above since we can precisely characterize some of their properties mathematically.

2. BB= Black box Methods

BB methods are characterized by random inputs or inputs collected from the environment. BB tests are cheap to generate since analysts need not reflect over the internal complexities of their systems. When designing BB tests, test engineers make little or no use of the internal details of a system. Two broad classes of BB methods are *approximate testing* and *sampling*. Test engineers often adjust the effort associated with testing using *stopping rules*. Approximate testing, sampling, and stopping rules are discussed below.

Over half this chapter discusses BB since, surprisingly, it turns out that the random dice of BB testing are an excellent cost-effective method of *detecting errors*. Random BB tests are essential for analyzing system reliability since it is good practice to test outside of the situations defined by the analysts [26, p670]. Randomized selection of

* *NOTE TO REVIEWERS*: looking at the proposed list of chapters for volume 2, it looks like other references should go here pointing to other volume 2 chapters.

Text	N tests	Text	N tests
[28]	4..5	[8]	≈ 6
[6]	5..10	[17]	8..10
[56]	10	[10]	< 13
[42]	40	[50]	50
[4]	200		

Figure 1: Number of tests proposed by different authors. Extended from a survey by [10].

test inputs may uncover errors that could be missed if testing is biased by the incorrect assumptions of the analysts [32].

Lest this chapter over emphasises BB, it is important to stress now that BB techniques are blind to the internal structure of a program. Once an error is *detected* using BB, then other methods such as WB or FM are required to *localize* the source of that error.

2.1. *Approximate Testing*

The author's experience in the Australian and American software industry is that the final version of a software system is often fielded after only a handful of tests (dozens to hundreds). A literature review strongly suggests that this experience is not atypical. For example, much of the expert systems literature proposes evaluations based on very few tests[†]; see Figure 1. Such small test sets can only ever be an *approximate test* of a system. For the rest of this section, we explore the case for approximate testing. We will find that in the majority of cases, approximate testing will suffice. However, a mathematical model of approximate testing suggests that approximate testing will fail at in at least 25% of cases. Hence, subsequent sections will try to improve on approximate testing.

Approximate testing assumes that:

1. A system can be sampled via a small number of inputs.
2. This small set of inputs might not be chosen with much care.

If our software contains pathways tangled like spaghetti, then these two assumption are clearly inappropriate. However, if software pathways are simple, then a few tests will adequately probe a system and approximate testing is an adequate strategy.

To see when approximate testing might work, consider the following example containing 15 binary variables $a, b \ldots, n, o$. Equation 1 tells us that this system needs $2^{15} = 32768$ tests. However, suppose that system has only one input and one output and pathways look like this:

$$\begin{array}{c} a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \\ input \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow output \\ k \rightarrow l \rightarrow m \rightarrow n \rightarrow o \end{array}$$

[†]Exception: [3] propose at least one test for every five rules and add that "having more test cases than rules would be best".

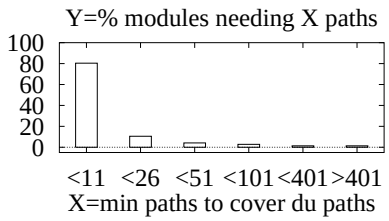

Figure 2: An analysis of hundreds of modules in a software system. 83% of the modules could be fully explored using less than ten tests. From [5]

Clearly, no tests are required for the top and bottom pathway since they are isolated from system inputs and outputs. Further, only a single test is required to cover the only pathway that connects inputs to outputs. If the paths in most programs look like this example, then software contains

1. A *small number of simple i/o pathways* from inputs to outputs, and
2. *Large unreachable regions*.

If so, then any input, selected at random, that can propagate through to any output will test a significant portion of the usable parts of the system. Hence:

- A small number of such inputs would hence be sufficient to test a system; and
- We can ignore Equation 1 and endorse approximate testing.

2.1.1. *Empirical Support for Approximate Testing*

There is much empirical evidence from the literature that real-world software comprises mostly simple i/o paths and large unreachable regions:

- Bieman & Schultz [5] 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*. Figure 2 shows their experimental results. At least for the system they studied, in the overwhelming majority of their modules, very few inputs exercised all the du-pathways.
- Harrold et.al. [29] studied how control-flow diagrams grow as program size grows. A worst-case control-flow graph is one where every program statement links to every other statement; i.e. the edges in graph grow with the square of the number of statements. However, for over 4000 Fortran routines and 3147 "C" functions, the control flow graph grows linearly with the number of statements. That is, at least in the systems seen in that study, the program pathways form single-parent trees and not complicated tangles.
- Colomb [14] compared the inputs presented to an medical expert system with its internal structure. Based on the number variables V and their states S, an

Program	% coverage			
	Block	Decision	p-use	c-use
TEX	85	72	53	48
AWK	70	59	48	55

Figure 3: Coverage reported by [32, p544]. "Block"= program blocks. "Decision"= program conditionals. "P-use"= pathways between where a variable is assigned and where it is used in a conditional. "C-use"= pathways between where a variable is assigned and where it is used, but not in a conditional.

analysis like Equation 1 made Colomb note that that system should have $S^V = 10^{14}$ internal states. However, after one year's operation, the inputs to that expert system only exercised 4000 states; i.e. in practice, this system only needed to handle a tiny fraction of the possible states ($4000 \ll 10^{14}$).

- Avritzer et.al. [2] studied the 857 different inputs seen in 355 days operation of an expert system. Massive overlap existed between these input sets. On average, the overlap between two randomly selected inputs was 52.9%. Further, a simple algorithm found that 26 carefully selected inputs covered 99% of the other inputs while 53 carefully selected inputs covered 99.9% of the other inputs.

- Horgan and Mathur [32] noted that testing often exhibits a *saturation* effect; i.e. most program paths get exercised early with little further improvement as testing continues. Saturation is consistent with programs containing large portions with simple shapes that are easily reached and other large portions that are so twisted in shape, that they will never be reachable. They report studies with the Unix report-generation language AWK [1]) and the word processor TEX [34]. Both AWK and TEX have been tested extensively for many years by their authors, with the assistance of a vast international user group. Elaborate test suites exist for those systems (e.g. [34]). Even after elaborate testing, large portions of TEX and AWK were not covered (see Figure 3).

2.1.2. *Theoretical Support for Approximate Testing*

If these case studies represented the general case, then we should have great confidence in the utility of approximate testing. The average shape of software can be inferred from the odds of reaching any part of the system from random input. If the odds are high, then the pathways to that part must be simple. To infer these odds, Menzies & Cukic [39] assumed that software had been transformed into a possibly cyclic directed graph containing and-nodes and or-nodes (e.g. Figure 4 would be converted to Figure 5). A simplified description of their analysis is presented here. For reasons of space, that simple description ignores certain details presented in the full description of the model [41] such as random variables, and testing for loops/contradictions.

To compute the odds of reaching some part of a program graph, we need tools. Our first tool is the standard *sampling-with-replacement* expression of Equation 2.

$$y = \quad 1 - \left((1 - x_j)^N\right) \tag{2}$$

```
procedure relax {
    if   tired=="no" AND weekend then {gotoMall; gotoParty;}
}
function weekend {
    return day=="saturday" OR day=="sunday"
}
procedure gotoMall {
    if   day=="sunday" then doThis
}
procedure gotoParty {
    atParty="yes"
     if   time>2am then  atParty="no"; gotoHome else gotoParty;
}
procedure gotoHome {
    doThat
}
```

Figure 4: A sample of procedural code.

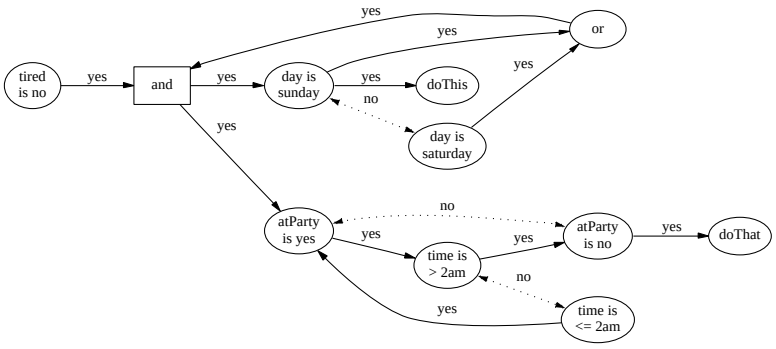

Figure 5: Conversion of the procedural code in Figure 4 to a graph containing no-edges (between incompatible nodes), and-nodes (which model conjunction), yes-edges (which model valid inferences), and or-nodes (which model disjunctions). And-nodes are shown as rectangles and or-nodes are shown as ellipses.

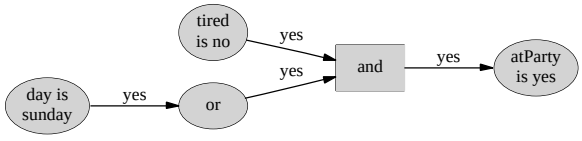

Figure 6: A program pathway extracted from Figure 5 that leads to “atParty=yes”. And-nodes are shown as rectangles and or-nodes are shown as ellipses.

To derive this expression, recall that an event with probability x does not happen after N trials with probability $(1-x)^N$. Hence, at probability y, the event will happen with the probability shown in Equation 2. This equation assumes test independence; i.e. the effects of performing one test do not affect the others.

Our second tool is an average case analysis of the *reachability* of programs. Assume that "in" number of inputs have been presented to a graph containing V nodes. From these inputs, we grow a tree of pathways down to some random node within the graph (e.g., see the shaded tree in Figure 6). The odds of reaching a node straight away from the inputs is $x_0 = \frac{in}{V}$. The probability of reaching an and-node with $andp$ parents is the probability of reaching all its parents; i.e. $x_{and} = x_i^{andp}$ where x_i is the probability we computed in the prior step of the simulation (and x_0 being the base case). The probability of reaching an or-node with orp parents is the probability of not missing any of its parents; i.e. $x_{or} = 1 - (1 - x_i)^{orp}$ (via Equation 2. If the ratio of and-nodes in a network is $andf$, then the ratio of or-nodes in the same network is $1 - andf$. The odds of reaching some random node x_j is the weighted sum of the probabilities of reaching and-nodes or or-nodes; i.e. $x_j = andf * x_{and} + orf * x_{or}$. We can convert x_j to the number of tests N required to be 99% sure of find a fault with probability x_j by rearranging Equation 2 to:

$$N = \frac{log(1-0.99)}{log(1-x_j)} \tag{3}$$

After 150,000 simulations of this model, the number of random inputs required to be 99% sure of reaching a node were usually either surprisingly small or impractically large:

- In 55% of the runs, less than 100 random tests had a 99% chance of reaching any node. This result is consistent with numerous simple i/o pathways.
- In 20% of the runs, the number of random tests required to be 99% sure of reaching any node was between one million and 10^{14}. This result is consistent with large unreachable regions.

In the remaining 25% of cases, systems needed between 10,000 and 1,000,000 random tests to be probed adequately. In these remaining cases, a few approximate tests would be inadequate to probe a system.

The good news from this simulation is that for most systems ($55 + 25 = 75\%$), a small number of tests will yield as much information as an impossibly large number of tests. For these systems:

- There is no point conducting lengthy and expensive testing since a limited testing regime will yield as much information as an elaborate testing procedure.
- Approximate testing is an adequate test regime.

The bad news from this simulation is twofold. Firstly, the Menzies & Cukic model is 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. Secondly, according to this model, approximate testing is inadequate in at least 25% of the space of systems explored by Menzies & Cukic. Hence, for those 25% of systems and for safety-critical systems, we need alternatives to approximate testing.

2.2. *Sampling*

One alternative to approximate testing is statistical sampling. Statistical sampling theory provides methods for assessing systems using far fewer tests than proposed by Equation 1. This section will discuss two examples of sampling. The first example will use *t-tests* and the second example will use the *sampling-with-replacement* equation.

The advantage of these sampling techniques is that unlike Equation 1, sampling techniques are not effected by the size of the system. The number of tests recommended by Equation 1 grows exponentially as the system size grows. However, the number of tests recommended by (e.g.) Equation 3 (see below) is only effected by required reliability.

The disadvantage of these sampling techniques was stated above. Once an error has been detected using cheap BB techniques, then other methods such as WB or FM may be required to localize and fix the error.

2.2.1. *Sampling Using T-Tests*

Using statistical t-tests, a surprisingly number of tests can certify even complex systems. This method will be presented by example.

A mere 40 tests was required to assess an expert system that controlled a complex chemical plant (125 kilometers of highly inter-connected piping) [42, 40]. The goal of that assessment was to test that the expert system was at least at good as human operators in running the plant. In that design, the expert system and the human operators took turns to run the plant. At the end of a statistically significant number of trials, the mean performance were compared using a t-test. Let m and n be the number of trials of expert system and the human experts respectively. Each trial generates a performance score (time till unusual operations): $X_1 \ldots X_m$ with mean μ_x for the humans; and performance scores $Y_1 \ldots Y_n$ with mean μ_y for the expert system. We need to find a Z value as follows:

$$Z = \frac{\mu_x - \mu_y}{\sqrt{\frac{S_x^2}{m} + \frac{S_y^2}{n}}} \quad \text{where} \quad S_x^2 = \frac{\Sigma(x_i - \mu_x)^2}{m-1} \quad \text{and} \quad S_y^2 = \frac{\Sigma(y_i - \mu_y)^2}{n-1}.$$

Let a be the degrees of freedom. If $n = m = 20$, the $a = n + m - 2 = 38$. We reject the hypothesis that expert system is worse than the human (i.e. $\mu_x < \mu_y$) with 95% confidence if Z is less than ($-t_{38,0.95} = -1.645$).

2.2.2. *Sampling With Replacement*

Another statistical technique is to use the sampling-with-replacement equation of Equation 3. In this approach, it is assumed that systems need only be tested to some pre-specified level of reliability; e.g. a probability of failure on demand of 10^{-9}. Further, it is assumed that tests are selected at random and a single test does not effect the results of the other tests. Given these assumptions, then we can use the sampling-with-

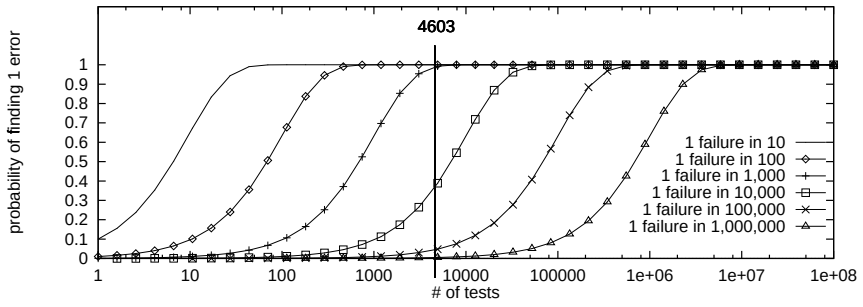

Figure 7: Chance of finding an error = $1-(1-failure\,rate)^{tests}$. Theoretically, 4603 tests are required to achieve a 99% chance of detecting moderately infrequent bugs; i.e. those which occur at a frequency of 1 in a thousand cases [27].

replacement expression of Equation 3 to assess the cost-benefit curve of random testing. Consider a search for a moderately low-frequency event such as one-in-a-thousand. For this system, $x_j = 0.001$ and Equation 3 tell us that 4603 randomly selected tests are required to be 99% certain that we will reveal that event. To compute other tests sizes using sampling-with-replacement, see Figure 7.

2.3. *Stopping Rules*

The discussion so far has assumed that the number of tests is somehow pre-specified and fixed. In practice, test engineers often use *stopping rules* to adjust the time spent in testing. Three such stopping rules are *reliability certification testing* and *bayesian stopping rules*, and *fault-based testing*.

2.3.1. *Reliability Certification Testing*

Certification tests using reliability demonstration charts have been introduced by Musa, Iannino and Okumoto [46]. They are based on sequential sampling theory, which is very efficient in the sense of giving the result (reliability certification) at the earliest possible time, i.e., smallest possible number of tests.

A *reliability demonstration chart* is shown in Figure 8. There are three regions on the chart: reject, test and accept. A failure is plotted to the chart when it occurs during random testing in which tests are selected according to the *operational profile* (an operational profile is a statement of what input values are expected at runtime). The vertical axis on the chart denotes the failure number, while the horizontal axis denotes normalized occurrence time (for example, occurrence time multiplied by the failure intensity objective). Depending on where the failure is plotted with respect to the graph regions, testing is stopped (with the program either accepted or rejected) or continued.

The number of tests required for reliability certification test in this technique depends on the position of the lines between reject, continue and accept regions. Their exact position will depend on

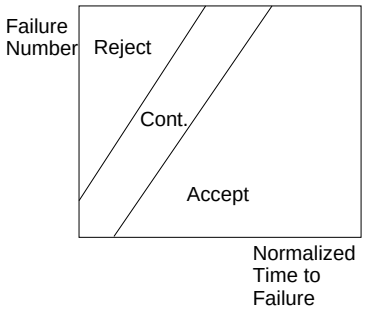

Figure 8: The reliability demonstration chart.

- The discrimination ratio, or the error in estimating failure intensity one is willing to accept.
- The consumer level of risk, or the probability one is willing to accept of falsely saying the failure intensity objective has been met when it is not.
- The supplier level risk, or the probability one is willing to accept of falsely saying the failure intensity objective has not been met when it is.

When risk levels and/or the discrimination ratio decrease, the continue region becomes larger. This situation requires more testing before reaching either accept or reject region.

This method is a practical version of a stopping rule, based on the required software reliability. It has seen its application in telecommunications industry. For the details of how to set appropriate levels of risks and discrimination ratio, interested readers are encouraged to look for statistical details in [47].

2.3.2. *Bayesian Stopping Rules*

When safety critical systems are tested, the usual goal is to achieve certain level of confidence that the predefined level of reliability has been achieved. Recall that Equation 2 said that $4,603$ successful tests are needed to have 99% confidence that the probability of failure is indeed less than or equal to 10^{-3}. However, this model does not address the question of what happens when failures occur *during* those 4,603 tests. Suppose a failure is detected at test 3,500. A common sense approach would require fixing the fault that caused the failure and the repetition of reliability certification, i.e., in this specific case, running all $4,603$ tests again. However, since failure occurrences in random testing are random events, debugging may not be necessary. In other words, program may actually be exhibiting the required reliability, even though it had failed.

The problem is how many additional tests need to be executed successfully following one or more failures, to be able to certify requested reliability without debugging.

Littlewood and Wright [36] proposed one solution, based on Bayesian statistics:

1. At the start of the certification test, compute n_1, the number of failure free executions for the certification test to succeed and stop. This figure might come from Equation 2.

2. Execute the test cases. If all n_1 executions succeed, stop testing and software reliability can be certified at the required level. Otherwise, a failure is observed at execution s_1 and testing stops.

3. In the light of evidence of one failure in s_1 executions, a number of further failure free executions, n_2, where $n_2 > n_1$, is determined.

4. Test executions proceed and either n_2 executions succeed (and reliability certified), of a failure is observed on demand $s_1 + s_2$. In the later case, steps 3 and 4 keep being repeated.

Note that if the program does not have the required reliability, testing may continue forever. In a sense, this technique is similar to Musa's reliability certification charts, but the reasoning that leads to accept or continue testing decisions is different.

Stopping rules proposed by Littlewood et. al. [37] are based on Bayesian statistics. Assume that target reliability level for a program is 10^{-3}, denoted by p_0. Testing begins with an ignorance prior probability of failure Θ, meaning that Θ is equally likely to assume any value between 0 and 1. As successful tests are executed, followed by observed failures, the prior information changes to reflect these observations. These observations are incorporated in a *Beta distribution* with appropriate values for its two parameters. Whenever a failure occurs, in addition to an update to prior distribution of failure probability, the additional number of successful tests that indicate a posterior distribution confirming the target failure probability, p_0, is computed. Readers interested in understanding the details of this approach are encouraged to read [36, 37, 16].

2.3.3. *Fault-Based Testing*

Software fault injection is the process of physically injecting a fault into the program. *Fault seeding* is a statistical fault injection method used to estimate the number of faults remaining in the program after the testing has been performed [44]. Let M denote the known number of injected faults, k the total number of faults detected through testing, and m the number of injected faults detected by testing ($m \leq k$ and $m \leq M$). Under the assumption that both injected and inherent faults are equally likely to be detected, an estimate of the number of inherent faults N is

$$N = \frac{M}{m}(k - m).$$

Stopping rules depend on the type of the program under test. Typical rules require that between 80% and 100% of the injected faults are detected by black box testing.

Mutation analysis is a technique similar to fault seeding [18]. Multiple copies of the program are made, and in each copy, one or more faults are injected using one of the mutant operators. Mutant program copies are tested with input data sets. The goal of mutation analysis is to determine which data set is able to detect the changes. Mutation analysis inspired Jeff Voas and his colleagues to estimate the conditional probability

that, given a fault in the program location, the data state becomes infected and results in incorrect outputs. Systematic generation and injection of faults into different program locations, combined with measurements of the above mentioned probability, is called *sensitivity analysis* [55]. Further, Voas declares that in order to achieve *testable software*, one needs to perform sensitivity analysis and go back to the specification and design phase and change the insensitive program locations, that is, those which are responsible for not propagating (injected) faults into failures. Alternatively, instead of redesign, more testing effort can be directed towards program segments with lower testability. No stopping rules have been defined for sensitivity analysis testing.

Nowadays it is generally agreed that fault based testing does not provide an insight into how good the code is per se. It rather answers "what if" scenarios by simulating human factor errors and environment failures. When safety is of concern, fault-based methods can be a useful complement to reliability assessment but cannot replace it. The main drawback of fault-based testing for the reliability assessment is the questionable (oversimplified) representativeness of injected faults.

3. WB= White Box Testing

WB partition testing is a testing regime where, unlike BB, analysts have information about where they should look to repair a failed test. In WB testing, analysts reflect over the internals of a program to invent test inputs that exercise K different *partitions* within a program. Partitions divide up the program's behavior into *equivalence classes*. Each class represents one interesting behavior of the system. For example, one equivalence class might relate to an incorrect password being offered at a login prompt. BB testing might test that prompt again and again with hundred of combinations of different character strings. The same test developed using WB methods might require only two tests: one for a correct login password and one for an incorrect password.

Since WB tests let us examine the internals of a program, they can be used to define stopping rules for a test regime. An often used criteria is *coverage*; i.e. stop testing a system when we have *covered* all parts of it. Various coverage criteria have been defined such as

- Exercise all lines in the program at least once.
- Check that all conditionals have been used at least once.
- For every conditional branch, ensure that both branches have been exercised.
- Ensure that all pathways between where a variable is set and where it is used are covered. This coverage criteria has been further divided according to how the variable is used; e.g. used in a conditional, used outside of a conditional.

For example, the Modified Condition Decision Coverage (MC/DC) criterion is very frequently by software testers in the aviation industry [51] for software based airborne systems. In this specific environment, a 100% coverage is required.

One of the benefits of WB should be that analysts can use their expertise into the process of finding errors. A common pre-experimental intuition is that this expertise greatly increases the chances that WB will find more errors than BB. However this

turns out not to be the usual case for three reasons: *incomplete coverage*, *partition creation* and *detection effectiveness* (discussed below). In general, WB is not better than BB at *detecting errors*. However, the real win with WB is that it is much better than BB at *locating errors* once they have been detected.

3.1. *Incomplete Coverage*

Coverage-based WB testing can be an inadequate testing strategy. Fenton [22, p302] reports that even when we try to explore the entire space of a program, the average reachable "objects" (paths, linearly independent paths, edges, statements) is only 40% at most. Some evidence for Fenton's claim was seen in Figure 3: even in mature, well-tested systems in frequent use, coverage may be far less than 100%.

Demanding full coverage can be prohibitively expensive. While MC/DC is less demanding than full branch condition coverage, it presents a huge cost overhead for large avionics systems. Boeing estimates that 40% of the software development costs for the 777 were spent on testing. Hence, testing to this standard has become a major cost driver in the development of new aircraft[‡].

Also, even if full coverage is achieved, coverage only comments on the structure of the code, and hence may not uncover problems associated with missing or incorrect requirements. Nor can coverage-based WB uncover systemic problems to do with the interaction between components. Other techniques such as BB are required to find these systemic problems.

3.2. *Problems with Partition Creation*

Creating the K partitions used in WB testing is a non-trivial task. An analyst must mentally consider how all inputs would flow into a system and past the program conditionals. If the program flows are pushed deep into the system, then the analyst will have a hard task ensuring that the flows are internally consistent. Automatic tools could be used to build the possible flows through a program.

Such automatic tools would face two problems. Firstly, they would have to execute over accurate representations of a system. Such a representation could come from either the specification documents or the actual system. If the specification is used, then experience strongly suggests that it will contain numerous inaccuracies that could confuse our automatic tools. If the actual system is used, then a call graph would have to be extracted from the code. Generating a correct call graph is problematic. For example, Murphy *et. al.* caution that in languages that support pointer to arbitrary constructs, then the problem is fundamentally intractable [45]. Different call graph generators tame this computational problem via a variety of heuristic design decisions. These heuristics alter the call graphs generated. For example, Murphy *et. al.* report significant differences in the graphs produced by different call graph generators [45].

The other problem with automatic partitioning is that it can be very slow. Gabow et.al. [24] showed that building pathways across programs with contradictions is *NP-complete* for all but the simplest software models (a software model is very simple if

[‡]The total development cost for the B777 was $5 billion. Approximately half of this was software development; hence roughly a billion dollars were spent on software testing.

it is very small, or it is a simple tree, or it has a dependency networks with out-degree $\leq$ 1). No fast and complete algorithm for NP-complete tasks has been discovered, despite decades of research.

3.3. *Detection Effectiveness*

The other problem with WB is *detection effectiveness*. The chances of detecting an error with WB probing is nearly the same as with BB [27]. This is a counter-intuitive result but it is simple to demonstrate. Let us assume that our K partitions each have a different probability x_k of detecting an error. Clearly, the chances of finding all the errors in all the partitions after N_k tests in each partition is

$$1 - \left(\prod_{k=1}^{K} (1 - x_k)^{N_k} \right) \tag{4}$$

WB can be compared to BB if we compare Equation 4 with the probability of finding an error with probability x_j after N random black box tests; i.e. Equation 2. To make the comparison meaningful, we should insist that the total number of tests performed is the same; i.e. $N = \sum_{k=1}^{K} N_K$. [27] performs a lengthy comparison of the ratio of Equation 4 to Equation 2 using various relationships between x_k and x_j. In the overwhelming majority of their studies, this ratio was nearly always very close to unity; i.e. WB was not much better than BB at detecting errors.

This bizarre and surprising result has been duplicated many times (see the literature reviews in [27, 26]). In only two cases has it has been refuted:

- In the special case where all inputs are equally likely, then it can be shown that WB using K partitions can be up to K times better than BB at finding errors [26]. However, given the high cost of creating the K partitions, a factor of K improvement in the utility of WB is not overly impressive.

- Suppose a programmer repeatedly comments out out half the remaining code until an error disappears. In effect, this programmer is performing a binary-chop partitioning strategy to create a partition with an increased chance of holding the error. In this case, x_k would increase to a value much larger than x_j and WB becomes a viable testing regime. That is, while WB may be not much better than BB for *detecting* errors, it is superior for *localizing* errors once they have been detected [27].

4. FM= Formal Methods

The picture emerging here is that WB augments BB methods. We saw above that BB methods can be surprisingly useful at detecting errors, but may give little assistance in solving the detected errors. On the other hand, the costs of WB may not be justified given its relatively weak error detection properties (compared to BB). Nevertheless, WB is better than BB at localizing the source of an error.

Extending the picture, we say that FM testing augments WB and BB methods. FM combines a powerful first-order query mechanism for detecting errors and

a method for finding the cause of the error. Theoretically, far fewer FM tests are required than with BB and WB, since a single FM first-order query is equivalent to many WB or BB test inputs [38]. Further, mature and highly optimized tools exist for FM testing.

As we shall see below, the benefits of FM come at considerable cost. Often only small critical sections of systems can be tested using FM. [38] argue that FM should be viewed as one method in a spectrum of testing regimes. Since formal methods can only be applied to a small part of a system, FM should be proceeded by cheaper forms of testing such as BB and WB to identify the important parts of a system.

4.1. *About* FM

In FM, we write a system *twice*[§]. Once a program or specification is understood, we write its again in a high-level formal representation. This representation contains the essential features of the specification. The representation is formal in the sense that all its constructs have a precise semantics which can be revealed by automatic *model checkers* such as SPIN [30].

A formal model has two parts: a *systems model* and a *properties model*. The systems model describes how the program can change the values of variables while the properties model describes global invariants that must be maintained when the system executes. Often, a temporal logic is used to express the properties model. Temporal logic is classical logic augmented with some temporal operators such as

$$\begin{aligned} \Box X: &\quad \text{always } X \text{ is true} \\ \Diamond X: &\quad \text{eventually } X \text{ is true,} \\ \bigcirc X: &\quad X \text{ is true at the next time point} \\ X \bigcup Y: &\quad X \text{ is true until } Y \text{ is true} \end{aligned}$$

For example, the simple pseudo-English requirement "the brake should always be applied between seeing the danger and the car stopping" might be written as the following properties model in temporal logic:

$$\begin{aligned} \Box((danger = seen \wedge !(car = stop) \ \wedge \Diamond \ (car = stop)) \\ \rightarrow (brake = on \ \cup \ (car = stop))) \end{aligned}$$

Modern model checkers search the systems model for a method of proving the negation of the properties model. If successful, then these model checkers can return a *counter-example* that describes exactly how the systems model can fail. Analysts find these counter-examples very useful in tracing out how the causes and fixed for a bug.

4.2. *The Costs of* FM

The three costs of FM are the *writing cost*, the *running cost*, and the *rewriting costs*. The writing cost has two components. Firstly, there is a short supply of analysts skilled in creating temporal logic models. Secondly, even when analysts with the right skills are available, the writing process is time-consuming. In recent years, much progress has been made in reducing this writing cost. For example:

- In the KAOS system [54], analysts write a properties model by incrementally augmenting object-oriented scenario diagrams with temporal logic statements.

[§] Exception: see F^2M, discussed below.

Potentially, this research reduces the costs of formal requirements analysis by integrating the writing of the properties model into the rest of the system development.

- Dwyer et.al. [19, 20] have identified *temporal logic patterns* within the temporal logic formulae seen in many real-world properties models. For each pattern, they have defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal logic formulae. In this way, analysts are shielded from the complexity of formal logics.

Another significant cost of FM is the *running cost* of model checking. A rigorous analysis of formal properties implies a full-scale search through the systems model; i.e. Equation 1. This space can be too large to explore, even on today's fast machines. Much of the research into FM focuses on how to reduce this running cost of model checking. Various techniques have been explored:

Abstraction or partial ordering: Only use the part of the space required for a particular proof. Implementations exploiting this technique can restrain how the space is traversed [25, 43], or constructed in the first place; e.g. [23, 52].

Clustering: Divide the systems model into sub-systems which can be reasoned about separately [13, 57, 11, 48].

Meta-knowledge: Avoid studying the entire space. Instead, only study succinct meta-knowledge of the space. One example used an eigenvector analysis of the long-term properties of the systems model under study [33].

Exploiting symmetry: Prove properties in some part of the systems model, then reuse those proofs if ever those parts are found elsewhere in the systems model [12].

Semantic minimization: Replace the space with some smaller, equivalent space [31] or ordered binary decision diagrams [7]. For example, the BANDERA system [15] reduces both the systems modeling cost and the execution cost via automatically extracting (slicing) the minimum portions of a JAVA program's bytecodes which are relevant to particular properties models.

While the above tools have all proved useful in their test domains, they may not be universally applicable:

- Certain optimizations require expensive pre-processing, such as [33].
- Exploiting symmetry is only useful if the system under study is highly symmetric.
- Clustering generally fails for tightly connected models.
- Splicing systems like BANDERA are very language specific. BANDERA only works on implemented JAVA systems and not for (e.g.) specification documents.

Due to the high running costs, a common cycle is to:

1. Write a formal model,
2. Try to run it,
3. Realize that it is too large to check formally,
4. Try to shrink the model it by rewriting it at some higher level of abstraction.

That is, apart from the *writing cost* and the *running cost*, the other cost of FM is the *rewriting cost*.

In summary, often only small descriptions of systems can be formally tested. Anecdotally, we know of one case where the invariants from 30 JAVA classes takes 1GB of main memory to check formally, even using a state-of-the-art automatic model checker. Testing larger systems may not be testable using FM since such larger systems would require exponentially more memory than 1GB. Consequently, in the general case, classic formal methods does not reduce the effort of testing a system. However, for the the kernel of mission-critical or safety-critical systems, the large cost of FM is often justified.

4.3. *Styles of Formal Methods*

The previous section described traditional FM. We denote this traditional style F^1M and distinguish it from other styles such as $F^{-1}M$ or *lightweight formal methods*; F^0M or *manual formal methods*; and F^2M or *full life cycle formal methods*. The index x in F^xM denotes the level of rigor and effort required to apply that style of formal methods testing.

One example of lightweight formal methods ($F^{-1}M$) is the work of Schnieder et.al. [52]. In this lightweight approach, a model checker was used to describe a system. However, only partial descriptions of the systems and properties models were constructed. Despite their incomplete nature, Schneider et.al. found that such partial models could still detect significant systems errors.

Leveson's work on software fault trees (SFT) [35] is an example of an ultra-lightweight formal method. SFTs have a fully formal semantics. Yet they are lightweight to construct since they are typically very small. For example, the SFT for an if-then-else statement has only a few entries since if-then-else is a simple construct. Leveson et.al. heuristically applied a library of SFTs to procedural code. They argue that SFTs found as many errors in less time than a traditional F^1M analysis. Note that this result endorses *either* the utility of $F^{-1}M$ *or* our above argument that approximate testing is often an adequate testing regime.

Manual formal methods (F^0M) implies the manual construction and manipulation of intricate mathematical descriptions of a systems model written in (e.g.) the Z notation. Due to the manual nature of F^0M, it can only be applied by highly skilled analysts to very small descriptions of programs.

F^1M was discussed in the previous section. F^1M can be criticized for being applied too late in the software life cycle. Such critics rhetorically ask

> "Why debug an incorrect system into correctness? Would it not be better to build demonstrably correct systems in the first place?"

Advocates of this *write-it-right* F^2M approach build systems from libraries of components. Automatic refinement methods specialize and combine members of the component library into an executable [53, 9]. Knowledge about the particular application being constructed is used to constrain and inform the refinement process.

The dream of F^2M is that systems will never need testing since there were generated from components that have been formally proved correct using an automatic refinement method that has also been proved correct. The reality of F^2M is that the technology currently available for automatic generation is not perfect. While refining a single component may maintain the correctness of that component, when components combine it is not clear that correctness can be guaranteed. Also, the cost of building and maintaining the library of formally proved components is non-trivial. Extensive and elaborate mathematical annotations must be added to each component in order to support proving it's correctness and combining it with other components. An open question for F^2M research is "will the benefits of F^2M be out-weighted by the cost of developing the component library?".

5. Summary

We have explored how many tests it takes to certify software. Tests have been characterized as samples of the space of possible pathways within a program. Such pathways clump together variables and any probe into that clump will yield information about the entire clump. Hence, the required number of tests need not be exponential on program size.

A system can be tested approximately when a few randomly selected probes finds most of the clumps. This does not happen when the internal pathways are too complex to condense into a small number of clumps. Based on a simulation of reaching nodes in an and-or graph, it was argued above that such overly-complex paths occur at least 25% of the time. Hence, approximate testing cannot be endorsed for safety critical and mission critical systems.

To improve on approximate testing, we then exploring sampling. Given a known required level of reliability, then Equation 2 can return the number of required random tests. For even moderate levels of reliability (e.g. find all faults with a frequency of one-in-a-thousand), then thousands of tests may be required (Equation 2 says that at $x_j = 10^{-3}$, 4603 random tests are required to be 99% certain of finding the fault).

When sampling, test engineers often use feedback from the test results to control when to stop testing. Three stopping criteria discussed here were reliability certification testing, bayesian stopping rules, and fault-based testing.

An alternative to random black box probing of a system is white box partitioning. White box testing allows an analyst to use knowledge of internal program structure to define test cases and stopping criteria based on system coverage. However, there any several drawbacks with white box testing including incomplete or expensive coverage, the cost of creating accurate partitions, and the comparative effectiveness of white box testing. Compared to random black box probing, white box testing is no better than *detecting errors* but is superior at *localizing errors*.

Formal methods can find errors with far fewer tests than white box or black box methods. Further, when an error is detected, formal method model checkers can re-

turn a counter-example showing exactly what must be changed to prevent the error. However, the benefits of formal methods come at considerable cost including the cost of writing the formal model (analysts skilled in formal analysis are in short supply), running the formal model (which, in the worst case, has to run through an exponential number of states in the program), and then rewriting the model if we need to reduce its size and associated runtime cost. Much research has been devoted to reducing these costs but, for the moment, formal methods can only be successfully applied to small descriptions of systems. Hence, it is good practice to precede formal methods with other, cheaper and less complete testing regimes, in order to focus the analysis on relevant portions.

This chapter reviewed different styles of formal methods. It is an open question if these different styles will change the way we use formal methods in the future. For example, full life cycle formal methods could remove the bugs before we insert them into our systems. However, this approach come at considerable cost. Lightweight formal methods are another exciting alternative style of formal methods. However, it is possible that "lightweight formal methods" are really just a synonym for "approximate testing"; i.e. any testing regime (lightweight formal methods or random black box testing) will quickly reveal many bugs.

Acknowledgements

Helen Burgess provided invaluable editorial advice for this article.

References

1. A.V. Aho, B.W. Kernigham, and P.J. Wienberger. *The AWK Programming Language*. Addison-Wesley, 1988.
2. A. Avritzer, J.P. Ros, and E.J. Weyuker. Reliability of Rule-Based Systems. *IEEE Software*, pages 76–82, September 1996.
3. A.T. Bahill, K. Bharathan, and R.F. Curlee. How the Testing Techniques for a Decision Support Systems Changed Over Nine Years. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(12):1535–1542, December 1995.
4. G. Betta, M. D'Apuzzo, and A. Pietrosanto. A Knowledge-Based Approach to Instrument Fault Detection and Isolation. *IEEE Transactions of Instrumentation and Measurement*, 44(6):1109–1016, December 1995.
5. J.M. Bieman and J.L. Schultz. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, 7(1):43–51, 1992.
6. D.G. Bobrow, S. Mittal, and M.J. Stefik. Expert Systems: Perils and Promise. *Communications of the ACM*, 29:880–894, 1986.
7. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3), September 1992.
8. B. Buchanan, D. Barstow, R. Bechtel, J. Bennet, W. Clancey, C. Kulikowski, T.M. Mitchell, and D.A. Waterman. *Building Expert Systems, F. Hayes-Roth and D. Waterman and D. Lenat (eds)*, chapter Constructing an Expert Sytem, pages 127–168. Addison-Wesley, 1983.
9. W. Buntime. Will Domain-Specific Code Syhthsis Become a Silver Bullet? *IEEE Intelligent Systems*, pages 9–15, March/April 1998.
10. J.P. Caraca-Valente, L. Gonzalez, J.L. Morant, and J. Pozas. Knowledge-based

Systems Validation: When to Stop Running Test Cases. *International Journal of Human-Computer Studies*, 2000. To appear.
11. D.J. Clancy and B.K. Kuipers. Model Decomposition and Simulation: A component based qualitative simulation algorithm. In *AAAI-97*, 1997.
12. E.M. Clark and T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. In *Fifth International Conference on Computer Aided Verification*. Springer-Verlag, 1993.
13. P. Clark and T. Ng. The CN2 Induction Algorithm. *Machine Learning*, 3:261–283, 1989.
14. R.M. 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`.
15. J. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasarenu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings ICSE2000, Limerick, Ireland*, pages 439–448, 2000.
16. B. Cukic and D. Chakrawarthy. Bayesian Framework for Reliability Assurance of a Deployed Safety Critical System. In *Proceedings of the 5th International Symposium on High Assurance Systems Engineering, Albuquerque, NM, November*, 2000.
17. P. Davies. Planning and Expert Systems. In *ECAI '94*, 1994.
18. R. DeMillo, R. Lipton, and F. Sayad. Hints on Test Data Selection: Help for the Practising Programmer. *IEEE Computer*, 11(4):34–41, April 1987.
19. M. B. Dwyer, G. S. Avrunin, and J.C. Corbett. A System Specification of Patterns. `http://www.cis.ksu.edu/santos/spec-patterns/`, 1997.
20. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *ICSE98: Proceedings of the 21st International Conference on Software Engineering*, May 1998.
21. N.S. Eickelmann and D.J. Richardson. An Evaluation of Software Test Environment Architectures. In *Proccesings, International Conference on Software Engineering*, pages 353–364, 1996.
22. N. E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.
23. M. Fujita. Model Checking: Its Basics and Reality. In *Asia and South Pacific - Design Automation Conference*, 1998.
24. H.N. Gabow, S.N. Maheshwari, and L. Osterweil. On Two Problems in the Generation of Program Test Paths. *IEEE Trans. Software Engrg*, SE-2:227–231, 1976.
25. P. Godefroid. On the Costs and Benefits of Using Partial-Order Methods for the Verificiation of Concurrent Systems (invited papers). In *The 1996 DIMACS workshop on Partial Order Methods in Verificaition, July 24-26, 1996*, pages 289–303, 1997.
26. W.J. Gutjhar. Partition vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, September/October 1999.
27. D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.
28. D. Harmon and D. King. *Expert Systems: Artificial Intelligence in Business*. John Wiley & Sons, 1983.
29. M.J. Harrold, J.A. Jones, and G. Rothermel. Empirical Studies of Control Dependence Graph Size for C Programs. *Empirical Software Engineering*, 3:203–211, 1998.
30. G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engi-*

neering, 23(5):279–295, May 1997.
31. G.J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States, 1999.
32. 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.
33. Y. Ishida. Using Global Properties for Qualitative Reasoning: A Qualitative System Theory. In *Proceedings of IJCAI '89*, pages 1174–1179., 1989.
34. D.E. Knuth. A Torture test for TEX. Technical Report STAN-CS-84-1027, Department of Computer Science, Stanford University, 1984.
35. N. Leveson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.
36. B. Littlewood and D. Wright. Stopping Rules for the Operational Testing of Safety Critical Software. In *Proceedings of the 25th Conference on Fault Tolerant Computing (FTCS 25), Pasadena, CA, July*, 1995.
37. B. Littlewood and D. Wright. Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software. *IEEE Transactions on Software Engineering*, 23(11):673–683, November 1997.
38. 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.
39. T. Menzies and B. Cukic. When to Test Less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://tim.menzies.com/pdf/00iesoft.pdf`.
40. Tim Menzies. Critical Success Metrics: Evaluation at the Business-Level. *International Journal of Human-Computer Studies, special issue on evaluation of KE techniques*, 51(4):783–799, October 1999. Available from `http://tim.menzies.com/pdf/99csm.pdf`.
41. Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Nondeterminate Systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.
42. T.J. Menzies. Evaluation Issues with Critical Success Metrics. In *Banff KA '98 workshop.*, 1998. Available from `http://tim.menzies.com/pdf/97langevl.pdf`.
43. T.J. Menzies and P. Compton. Applications of Abduction: Hypothesis Testing of Neuroendocrinological Qualitative Compartmental Models. *Artificial Intelligence in Medicine*, 10:145–175, 1997. Available from `http://tim.menzies.com/pdf/96aim.pdf`.
44. H. Mills. *Software Productivity*. Little, Brown, 1983.
45. G.C. Murphy, D. Notkin, and E.S.C. Lan. An Empirical Study of Static Call Graph Extractors. Technical Report TR95-8-01, Department of Computer Science & Engineering, University of Washington, 1995.
46. J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw Hill, 1987.
47. John Musa. *Software Reliability Engineered Testing*. McGraw-Hill, 1998.
48. K.M. Olender and L.J. Osterweil. Interprocedural Static Analysis of Sequencing Constraints. *TOSEM*, 1(2):21–52, 1992.
49. A.D. Preece and R. Shinghal. Verifying Knowledge Bases by Anomaly Detection: An Experience Report. In *ECAI '92*, 1992.
50. C. Loggia Ramsey and V.R. Basili. An Evaluation for Expert Systems for Software Engineering Management. *IEEE Transactions on Software Engineering*, 15:747–759, 1989.

51. INC RTCA. RTCA DO178B: Software Considerations in Airborned Systems and Equipment Consideration, 1992. December 1.
52. F. Schneider, S.M. Easterbrook, J.R. Callahan, G.J. Holzmann, W.K. Reinholtz, A. Ko, and M. Shahabuddin. Validating Requirements for Fault Tolerant Systems using Model Checking. In *3rd IEEE International Conference On Requirements Engineering*, 1998.
53. D. R. Smith. KIDS: A Semi-Automated Program Development System. *IEEE Transactions on Software Engineering (SE), Sept*, 16(9), 1990.
54. A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, November 1998.
55. J.M. Voas and K.W. Miller. Software Testability: The New Verification. *IEEE Software*, pages 17–28, May 1995.
56. V.L. Yu, L.M. Fagan, S.M. Wraith, W.J. Clancey, A.C. Scott, J.F. Hanigan, R.L. Blum, B.G. Buchanan, and S.N. Cohen. Antimicrobial Selection by a Computer: a Blinded Evaluation by Infectious Disease Experts. *Journal of American Medical Association*, 242:1279–1282, 1979.
57. Z. Zhang. An Approach to Hierarchy Model Checking via Evaluating CTL Hierarchically. In *Fourth Asian Test Symposium*, 1995.