An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity*

Jeffrey M. Voas
RST Corp.
11150 Sunset Hills Road
Suite 250
Reston, VA 22090 USA
(703) 742-8873
jmvoas@isse.gmu.edu

Keith W. Miller
Dept. of Computer Science
Sangamon State University
Springfield, IL USA
(217) 786-6770

Jeffery E. Payne
RST Corp.
11150 Sunset Hills Road
Suite 250
Reston, VA 22090 USA
(703) 742-8873

Abstract

This paper compares the dynamic testability prediction technique termed "sensitivity analysis" to the static testability technique termed cyclomatic complexity. The application that we chose in this empirical study is a CASE generated version of a B-737 autoland system. For the B-737 system we analyzed, we isolated those functions that we predict are more prone to hide errors during system/reliability testing. We also analyzed the code with several other well-known static metrics. This paper compares and contrasts the results of sensitivity analysis to the results of the static metrics.

I. Introduction

The adage that non-exhaustive software testing cannot reveal the absence of errors and only their existence is as true today as it was when Dijkstra wrote it [4, 1]. Unfortunately, between the time then and now, we have begun to build orders-of-magnitude more complex systems while our testing technologies are no more advanced. Thus the same problems that we had in past years when testing a 1000 line program are compounded when we apply those techniques to a 10M line program today.

We must admit that we are building software systems that are destined to be inadequately tested. Since we know this *a priori*, it suggests that we should look for techniques that aid the testing process where the process is known to be weak. In this paper, we discuss one such technique: a method that quantifies the dynamic testability of a system that is undergoing system/reliability testing. We will then compare the results of this technique to other metrics that are in wide-spread use today.

Software testing is performed for generally two reasons: (1) detect faults so that they can be fixed, and (2) reliability estimation. The goal of the dynamic testability measurement technique presented in this paper is to strengthen the software testing process as it applies to reliability estimation. Dynamic testability analysis is less concerned with fault detection, even though it is plausible that a function that is more likely to hide faults during system testing may also be more likely to hide faults during unit testing. Instead, dynamic testability analysis is concerned with a *lack* of fault detection.

II. Static Software Metrics

The study of software metrics has grown out of a need to be able to express quantitative properties about programs. The first software metric was simply a count of the number of lines. This was acceptable as a way of measuring program size, but was not applicable to other software characteristics.

Software complexity is another metric that tries to relate how difficult a program is to understand. In general, the more difficult, the more likely that errors will be introduced, and hence the more testing that will be required. Thus it is common for developers to relate a software complexity measurement to the allocation of testing resources. It is our experience, however, that software complexity is still too coarse-grained of a metric to relate to the testing of *critical software systems*, those that must fail less than once in 10^9 executions (or some other large number). Thus even though software complexity can be useful as a first-stab at how much testing to perform and where, it is too coarse for assessing reliability in the ultra-reliable region of the input space.

In this paper, we have considered 6 software metrics that are available in the PC-METRIC 4.0 toolkit: (1) Software Science Length (N) (2) Estimated Software Science Length ($N^\wedge$), (3) Software Science Volume (V), (4) Software Science Effort (E), (5) Cyclomatic Complexity (VG1), and (6) Extended Cyclomatic Complexity (VG2). We will briefly mention what these metrics are; in general, any software engineering text will go into more depth on these metrics for the inquisitive reader.

Halstead [2] observed that all programs are comprised of operators and operands. He defined N_1 to be the number of total operators and N_2 to be the number of total operands. He defined length of a program, N, to be:

$$N = N_1 + N_2.$$

Halstead also has a predicted length metric, $N^\wedge$, that is given by:

$$N^\wedge = n_1 \cdot log_2(n_1) + n_2 \cdot log_2(n_2),$$

where n_1 is the number of unique operators and n_2 is the number of unique operands. Halstead has another metric that he terms volume, V, that is given by:

$$V = N \cdot log_2(n_1 + n_2).$$

Halstead's Effort metric, E, is given by:

$$E = V/L,$$

however most researchers use [5]:

$$E = V/(2/n_1 \cdot n_2/N_2)$$

McCabe's cyclomatic complexity metric is less based on program size (as are Halstead's measures) and more on information/control flow:

$$V(g) = e - n + 2$$

where n is the number of nodes in the graph and e is the number of edges, or lines connecting each node. It is the cyclomatic complexity metric that we are more interested in for this paper, and most importantly how cyclomatic complexity compares to the dynamic testability measure presented in Section 3.

III. Testability Analysis

Software testability analysis measures the benefit provided by a software testing scheme to a particular program. There are different ways to define the "benefit" of tests and testing schemes, and each different definition requires a different perspective on what testability analysis produces. For instance, software testability has sometimes been referred to as the ease with which inputs can be selected to satisfy structural testing criteria (e.g., *statement coverage*) with a given program. With this perspective, if it were extremely difficult to find inputs that satisfied a structural coverage criteria for a given source program, then that program is said to have "low testability" with respect to that coverage criteria. Another view of software testability defines it as a prediction of the probability that existing faults will be detected during testing given some input selection criteria C. Here, software testability is not regarded as an assessment of the difficulty to select inputs that cover software structure, but more generally as a way of predicting whether a program would reveal existing faults during testing according to C.

In either definition, software testability analysis is a function of a (*program, input selection criteria*) pair. Different input selection criteria choose test cases differently: inputs may be selected in a random black-box manner, their selection may be dependent upon the structure of the program, or their selection may be based upon other data or they may be based on the intuition of the tester. Testability analysis is more than an assertion about a program, but rather is an assertion about the ability of an input selection criteria (in combination with the program) to satisfy a particular testing goal. The same syntactic program may have different testabilities when presented with different input selection criteria.

In order for software to be assessed as having a "greater" testability by the semantic-based definition, it must be likely that failure occurs if a fault were to exist. To understand this likelihood, it is necessary to understand the sequence of events that lead to software failure. (By software failure, we mean an incorrect output that was caused by a flaw in the program, not an incorrect output caused by a problem with the environment or input on which the program is executing.) Software failure only occurs when the following three conditions occur in the following sequence:

1. A input must cause a fault to be *executed.*
2. Once the fault is executed, the succeeding data state must contain a *data state error.*

3. After a data state error is created, the data state error must ***propagate*** to an output state.

The semantic-based definition of testability predicts the probability that tests will uncover faults if a fault exists. The software has high testability for a set of tests if the tests are likely to detect any faults that exist; the software has low testability for those tests if the tests are unlikely to detect any faults that exist. Since it is a probability, testability is bounded in a closed interval [0,1]. In order to make a prediction about the probability that existing faults will be detected during testing, a testability analysis technique should be able to quantify (meaning predict) whether a fault will be executed, whether it will *infect* the succeeding data state creating a data state error, and whether the data state error will propagate its incorrectness into an output variable. When all of the data state errors that are created during an execution do not propagate, the existence of the fault that trigged the data state errors remains hidden, resulting in a lower software testability.

Software sensitivity analysis is a code-based technique based on the semantic definition of testability; it injects instrumentation that contains program mutation, data state mutation, and repeated executions to predict a minimum non-zero fault size [7, 13]. The minimum non-zero fault size is the smallest probability of failure likely to be induced by a programming error based upon the results of the injected instrumentation. Sensitivity analysis is not a testing technique, and thus it does not use an oracle, and can be completely automated (provided that the user initially tells the technique where in the code to apply the analysis).

Software sensitivity analysis is based on approximating the three conditions that must occur before a program can fail: (1) execution of a software fault, (2) creation of an incorrect data state, and (3) propagation of this incorrect data state to a discernible output. This three part model of software failure [9, 10] has been explored by others, but not in the manner in which sensitivity analysis explores it. In this paper we examine how to apply sensitivity analysis to the task of finding a realistic minimum probability of failure prediction when random testing has discovered no errors.

In the rest of this section we give a brief outline of the three processes of sensitivity analysis. To simplify explanations, we will describe each process separately, but in a production analysis system, execution of the processes would overlap. As with the analysis of random testing, the accuracy of the sensitivity analysis depends in part on a good description of the input distribution that will drive the software when operational (and when tested).

Before a fault can cause a program to failure, the fault must be executed. In this methodology, we concentrate on faults that can be isolated to a single *location* in a program. This is done because of the combinatorial explosion that would occur if we considered distributed faults. A location is defined as a single high level language statement. Our experiments to date have defined a location as a piece of source code that can change the data state (including input and output files and the program counter). Thus an assignment statement, **if**, and *while* statement define a location. The probability of execution for each location is determined by repeated executions of the code with inputs selected at random from the input distribution. An automated testability system, *PiSCES* [11], controls the instrumentation and bookkeeping.

If a location contains a fault, and if the location is executed, the data state after the fault may or may not be changed adversely by the fault. If the fault does change the data state into an incorrect data state, we say the data state is *infected*. To estimate the probability of infection, the second process of sensitivity analysis performs a series of syntactic mutations on each location. After each mutation, the program is re-executed with random inputs; each time the location is executed, the data state is immediately compared with the data state of the original (unmutated) program at that same point in the execution. If the internal state differs, infection has taken place. And this is recorded by *PiSCES* and reported back to the user.

The third process of the analysis estimates propagation. Again the location is monitored during random tests. After the location is executed, the resulting data state is forcefully changed by assigning a random value to one data item using a predetermined internal state distribution. After the internal data state is changed, the program continues executing until an output results. The output that results from the changed data state is compared to the output that would have resulted without the change. If the outputs differ, propagation has occurred and *PiSCES* reports that back to the user as a probability estimate.

For a test case to reveal a fault, execution, infection, and propagation must occur; without these three events occurring, the execution will not result in failure. And for a specific fault, the product of the probability of these events occurring is the actual probability of failure for that fault. Each sensitivity analysis process produces a probability estimate based on the number of trials divided by the number of events (either execution, infection, or propagation). The product of these estimates yields an estimate of the probability of failure that would result when this location contains a fault. Since we are approximating the model of how faults result in failures, we also take this multiplication approach when we predict the minimum fault size and multiply the minimum infection estimate, minimum propagation estimate, and execution estimate for a given location. This produces the testability of that location. We then take the location with the lowest non-zero testability to be the testability of the overall program.

IV. PiSCES

Several proof-of-concept sensitivity analysis prototypes were built in the early 1990s. *PiSCES* is the commercial software testability tool that evolved from these prototypes. *PiSCES* is written in C++ and operates on programs written in C. The recommended platform for *PiSCES* is a Sparc-2 with 16 mbytes of memory, 32 mbytes of swap space, and 20 mbytes of hard disk space. For larger C applications, the amount of memory that *PiSCES* needs increases, and thus we currently are limited to running around 3,000-4,000 lines of source code at a time through *PiSCES*. For larger systems, we perform analysis on a part of the code, and when that is done, we perform analysis on another part until all of the code has received dynamic testability analysis. This "modular approach" is how we get results for systems larger than 4,000 SLOC.

PiSCES produces testability predictions by creating an "instrumented" copy of your program and then compiling and executing the instrumented copy. Although it is hard to determine precisely, given the default settings that *PiSCES* uses, the instrumented version of your program is approximately 10 times as large as the original source code. The instrumented copy is then executed with inputs that are either supplied in a file or *PiSCES* uses random distributions from which it generates inputs.

V. Dynamic Testability Results

We were supplied with a C version of a B-737 autopilot/autoland that had been generated by a CASE tool; the CASE tool has been under development by NASA-Langley and one of their vendors for several years. We were told that as far as NASA knew, this version of the autopilot/autoland had never failed; it appears to be a correct version of the specification. The version consisted of 58 functions; parameters to the system included information such as direction of wind, wind speed, and speed of gusts. The version we used is not embedded in any commercial aircraft. Instead, the version is based on the specification of the system that is embedded on aircraft, and hence this code should contain most (if not all) of the functionality of the production aircraft system.

We should mention that the B737 source code was approximately 3000 SLOCs, and it represents the largest program to receive sensitivity analysis in its entirety to date. Our results here are based on 2000 randomly generated input cases that are correlated to the following types of landing conditions:

1. no winds at all,
2. moderate winds, and
3. extremely strong winds with high gusts.

(We think it was important to exercise three major classes of scenarios that the system would encounter in operation.) We should mention that we found similar results [12] when we used a different test suite with 1000 randomly generated inputs. The total amount of clock time that it took for *PiSCES* analysis to run and produce the results was 55 hours on a Sparc-2 (there were no other major jobs running on that platform during this time).

According to the Squeeze-Play model for testing sufficiency for the B-737 version, sensitivity analysis recommends 11,982,927 system level tests [8]. This is based on the conservative testability prediction of $< 2.5E - 07$ for the entire program; ***a conservative testability translates into a liberal estimate of the amount of needed testing.*** We use a conservative testability to ensure that we are not fooled into believing that we have done enough testing when we really have not. A testability written as an inequality indicates that *PiSCES* encountered at least one location that did not execute or propagate during analysis. One possible quantification of this situation is to assign a testability of 0.0, but that creates problems for further analysis. Instead, *PiSCES* makes a reasonable estimate on testability and signals the singularity with the inequality. The process for doing this as well as the mathematics are described in the *PiSCES Version 1.0 User's Manual*. (A 0.0 testability produces an infinite amount of testing needed which is useless to testers.) Since testing on approximately 12 million inputs is impractical, there are other alternatives for increasing the testability; if these alternatives are applied successfully, they will decrease the number of tests required, but we will not explain here how that is done.

We now show the results for the 58 individual functions of the B737.c system in Figures 1 and 2. As you can see, there are 15 functions out of the 58 that have a testability of greater than 0.01. These are functions that the developer/tester need not worry over; they appear to have little fault hiding ability. This information also tells the developer which functions (the other 43) are more worrisome (in terms of hiding faults at the system level of testing); by immediately isolating those functions of low testability, we gain insight as to where additional testing resources are needed. Note that the degree to which we consider a function to be "worrisome" is a function of how much testing is considered feasible.

As you can see from the bar charts, there were many functions of low testability. This does not say that these functions are incorrect (recall that this program has never failed for NASA), but rather that these functions should receive special consideration during V&V. In our tool, there are ways of decreasing the recommended testing costs if the user knows that the regions of the code where the low testabilities occur are not hiding faults. Although such knowledge is difficult to obtain, it does provide the user with a justifiable way of performing code inspections and testing sufficiency.

VI. Static Metric Results

This section provides several sets of data that we collected from the B737 source code when we ran it through a commercial metrics package with the default settings. Table 1 and Table 2 display the results that were attained by running PC-METRIC 4.0 [5].

VII. Comparison of Results

Some software researchers and practitioners have equated testability with McCabe's cyclomatic complexity or some other static metric. We contend that such static measures do not capture the dynamic, data dependent nature that is fundamental to testing and our analysis of the effectiveness of testing.

In 1990, we introduced both a new definition of testability and a new method for measuring testability based on our definition. Still, we are frequently asked how our definition compares to cyclomatic complexity, which we feel is a valid question. In this section, we will try to show how these two measurement methods differ, and what these differences mean for the typical tester or QA manager.

As we have shown in Section 5, the B737 code had functions of high testability and lower testability. If the reader then considers Table 1, we immediately see that the VG1 values for the functions of B737 never exceeded 7, and for VG2, the functions never exceeded 10. According to the cyclomatic complexity measures, all of these functions are labeled as "not complex;" however sensitivity analysis has found that many of these functions are more likely to hide faults during testing than McCabe's numbers might suggest.

Our interpretation for why this is true is simply how the two metrics view a program; sensitivity analysis is based on the semantic meaning of the program, whereas cyclomatic complexity is based on an abstract and structural view of the program. It is true that the structural view has some impact on the semantics of the program, however during system level testing, we argue that the information provided by cyclomatic complexity is essentially useless in terms of how much testing to perform. Thus we conclude that for unit testing, cyclomatic complexity is an easy means of attaining a feeling for how good the structure of the program is (essentially as a "spaghetti" code type of measure), however for critical systems, we contend that the semantic perspective on testability provided by sensitivity analysis is far more valuable. Sensitivity analysis costs more, but the value added is also increased.

VIII. Conclusions

We contend that the preliminary results of experiments in software sensitivity are sufficient to motivate additional research into quantifying sensitivity analysis [13, 6]. Not only do we think that this technique may hold promise in assessing critical systems, but in Hamlet's award winning *IEEE Software* paper [3] and The National Institute of Standards and Technology's report on software error analysis [14], sensitivity analysis is acknowledged as a technique that should be further explored for its potentially enormous impact on assessing ultra-reliable software.

Although the subprocesses of sensitivity analysis will in all likelihood require minor revisions as more is learned about fault-based analysis, the ideas that motivate sensitivity analysis dispute the contention that software testing is the only method of experimentally

Procedure	N	$N^\wedge$	V	E	VG1	VG2
LIMITER	37	46	145	2186	3	3
LIM_180	59	72	259	4001	3	3
ONED	137	156	714	25760	3	7
INTEGRATE	45	57	188	1642	3	3
FOLAG	43	68	186	2243	2	2
WASHOUT	53	72	233	3215	2	2
STATE	17	24	56	329	1	1
EZSWITCH	62	113	301	4694	2	3
KOUNT	66	71	290	3508	4	4
MODLAG	130	185	701	16964	2	3
MODLAG_1	108	76	482	7706	7	7
MODLAG_2	35	64	149	1772	2	3
MODLAG_3	68	82	308	5263	3	3
DZONE	22	40	84	603	1	1
AUTOPILOT	276	610	1842	25716	2	2
MODE	172	323	1016	5166	1	1
MODES_2	178	351	1072	7309	1	1
MODE1	139	209	763	6940	1	10
MODE2	120	210	663	8948	1	8
CALC_GSTRK	120	134	605	9837	6	8
MODE3	109	168	576	7615	1	7
THROT	73	144	368	1432	1	1
ATHROT	180	310	1072	21174	5	7
PRE_FLARE	79	168	418	4840	2	2
VER_SINE	29	57	121	605	1	1
WIND_SHEAR	99	151	512	6763	1	1
SPEEDC	18	36	67	320	1	1
AFTLIM	58	97	273	3926	3	6
EPR_GAIN	43	72	189	2302	2	2
LONG_x	111	242	620	2391	1	1
LONGAP	226	399	1408	28817	3	3
CALC_HR	15	28	52	227	1	1
FLARE_CONTROL	135	185	728	12557	5	5
CALC_HDER	39	72	171	1485	2	2
PRE_FLARE_LONG	98	185	528	7332	2	2
IN_ON_BEAM	39	71	171	1225	2	2
GSE_ADJ	37	67	160	1270	1	1
BEFORE_GSE	51	96	240	2657	1	2
BANK_ADJ	17	33	61	457	1	1
PITCH_ADJ	39	81	176	1491	2	2
CAS_ADJ	25	53	102	657	1	1
LATERAL	115	270	656	2394	1	1
LATAP	192	362	1173	19388	2	2
LOC_ERROR	78	128	390	4095	2	2
CROSS_VEL	18	36	67	320	1	1
LOCCMD	76	128	380	5510	2	2
LOCINT	46	101	219	1884	1	2
LOCCF	78	117	383	4521	4	4
CROSSTKADJ	33	71	145	957	2	2
BANK	90	156	469	5894	3	4
PHICMDFB	83	145	426	5830	2	3
RtoA_XFD	103	191	559	7847	2	2
CALC_PSILIM	57	76	254	2898	3	3
SPOILER	51	81	231	2391	2	2
AIL_CMD	97	169	513	5287	4	4
RUDDER_CMD	75	145	385	3751	3	3
OUTERLOOPS	39	69	169	650	1	1
AUTOOL	69	117	339	3386	3	4

Table 1: Software Science Length (N) Estimated Software Science Length ($N^\wedge$), Software Science Volume (V), Software Science Effort (E), Cyclomatic Complexity (VG1), and Extended Cyclomatic Complexity (VG2) for B737.c

Metric	Score
Software Science Length (N):	4707
Estimated Software Science Length ($N^{\wedge}$):	4107
Software Science Volume (V):	42147
Software Science Effort (E):	7963380
Estimated Errors using Software Science ($B^{\wedge}$):	13
Estimated Time to Develop, in hours ($T^{\wedge}$):	123
Cyclomatic Complexity (VG1):	70
Extended Cyclomatic Complexity (VG2):	111
Average Cyclomatic Complexity:	1
Average Extended Cyclomatic Complexity:	1
Average of Nesting Depth:	1
Average of Average Nesting Depth:	0
Lines of Code (LOC):	3312
Physical Source Stmts (PSS):	2683
Logical Source Stmts (LSS):	569
Nonexecutable Statements:	861
Compiler Directives:	9
Number of Comment Lines:	1384
Number of Comment Words:	1985
Number of Blank Lines:	629
Number of Procedures/Functions:	58

Table 2: Summary of Static Metric Scores

quantifying software reliability. We believe that dynamic testability analysis is a new form of software validation, because it is quantifying a semantic characteristic of programs. We cannot guarantee that sensitivity analysis will assess reliability to the precisions required for life-critical avionics software, because as we have pointed out, low testability code can never be tested to any threshold that would strongly suggest that faults are not hiding. However, we do think it is premature to declare such an assessment impossible for all systems, and we feel that this topic deserves attention both from the avionics community as well as the software engineering and testing communities.

This experiment demonstrates important differences between static and dynamic analysis of how much testing is required. Admittedly, dynamic information is far more expensive to attain; but for the additional cost, the precision derived we feel is justified. This expense comes mainly from the fact that the input space and probability density function are also considered when assessing how much testing is necessary, not only the structure of the code. And this expense is in computer time, not human time.

We have felt that static software metrics are too assumption-based to be useful for predicting how to test critical systems. For this reason, we developed a new perspective on testability, a new way of measuring that definition, and commercialized a tool to perform the measurement.

Acknowledgement

This experiment was funded through NASA-Langley Grant NAG-1-884. The authors thank Carrie Walker for supplying us with the B737 auto-generated code from the ASTER tool.

Disclaimer

The code supplied to us was from NASA and not Boeing, and as far as we know, this code and the testability results do not reflect the quality of the software used in Boeing aircraft or produced by Boeing. Boeing is in no way affiliated with this experiment nor RST Corporation.

References

[1] E. Dijkstra. Structured Programming. In *Software Engineering, Concepts, and Techniques.* Van Nostrand Reinhold, 1976.

[2] M. H. Halstead. *Elements of Software Science.* New York:Elsevier North-Holland, 1977.

[3] D. Hamlet. Are We Testing for True Reliability? *IEEE Software*, pages 21–27, July 1992.

[4] O. J. Dahl, E. W. Dijsktra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[5] Set Laboratories Inc. PC-METRIC User's Manual.

[6] J. Voas, J. Payne, C. Michael and K. Miller. Experimental Evidence of Sensitivity Analysis Predicting Minimum Failure Probabilities. In *Proc. of COMPASS'93.*, NIST, Washington DC, June 1993.

[7] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March 1991.

[8] J. Voas and K. Miller. Improving the Software Development Process Using Testability Research. In *Proc. of the 3rd Int'l. Symposium on Software Reliability Engineering.*, pages 114–121, Research Triangle Park, NC, October 1992. IEEE Computer Society.

[9] L. J. Morell. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.

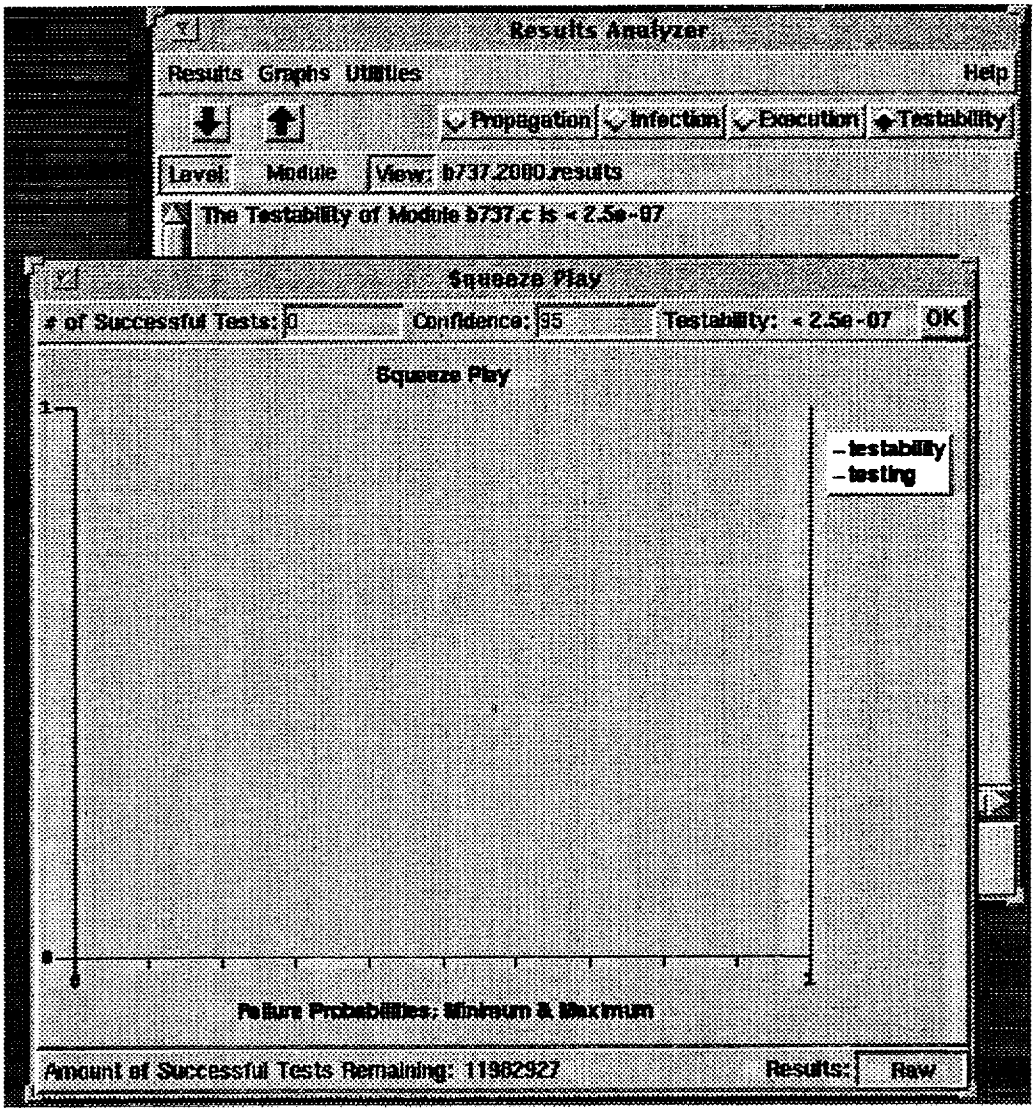
[10] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16, August 1990.

[11] J. Voas, K. Miller, and J. Payne. PISCES: A Tool for Predicting Software Testability. In *Proc. of the Symp. on Assessment of Quality Software Development Tools*, pages 297–309, New Orleans, LA, May 1992. IEEE Computer Society TCSE.

[12] J. Voas, K. Miller, and J. Payne. Software Testability and Its Application to Avionics Software. In *Proc. of the 9th AIAA Computers in Aerospace*, San Diego CA, October 19-21 1993.

[13] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

[14] W. Peng and D. Wallace. Software Error Analysis. Technical Report NIST Special Publication 500-209, National Institute of Standards and Technology, Gaithersburg, MD, April 1993.



Level: Module

View: b737.2080.results

The Testability of Module b737.c is < 2.5e-07

# of Successful Tests: 0

Confidence: 95

Testability: < 2.5e-07

Amount of Successful Tests Remaining: 11982927

Results: Raw

Figure 1: Testing needed given "raw" testability score for autoland module.

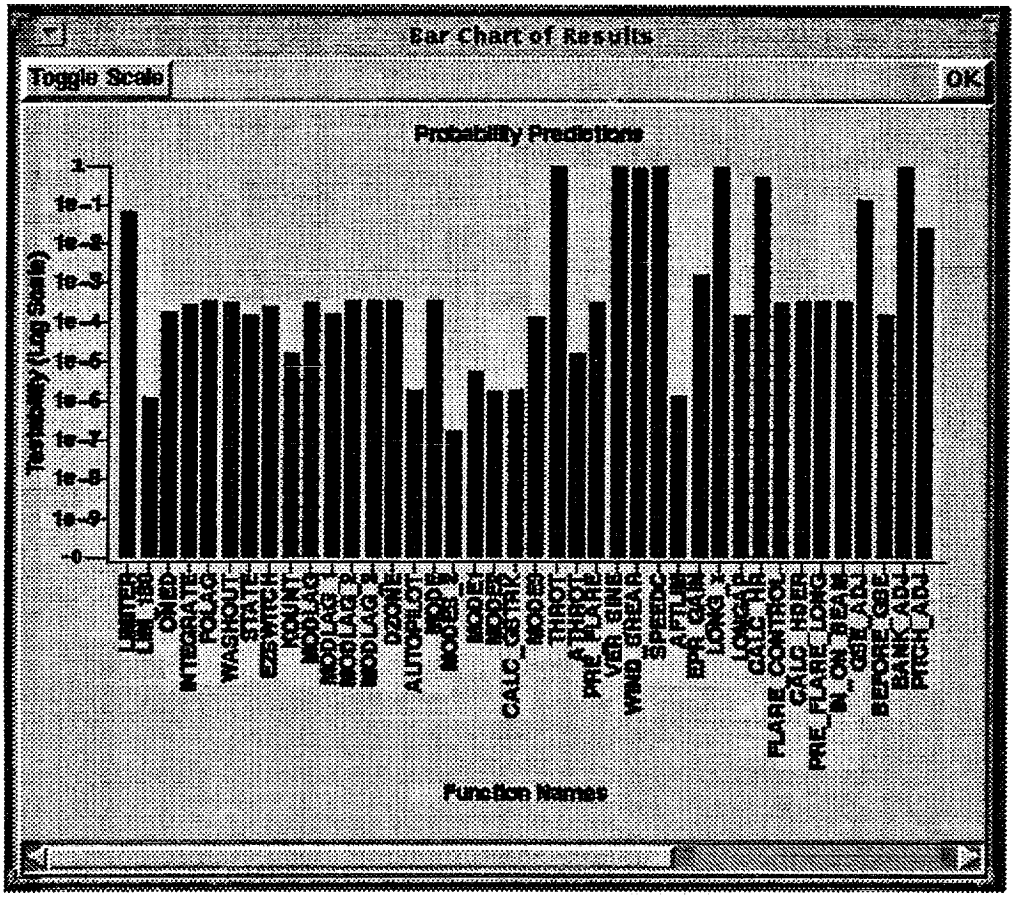


Figure 2: Function testabilities for autoland.

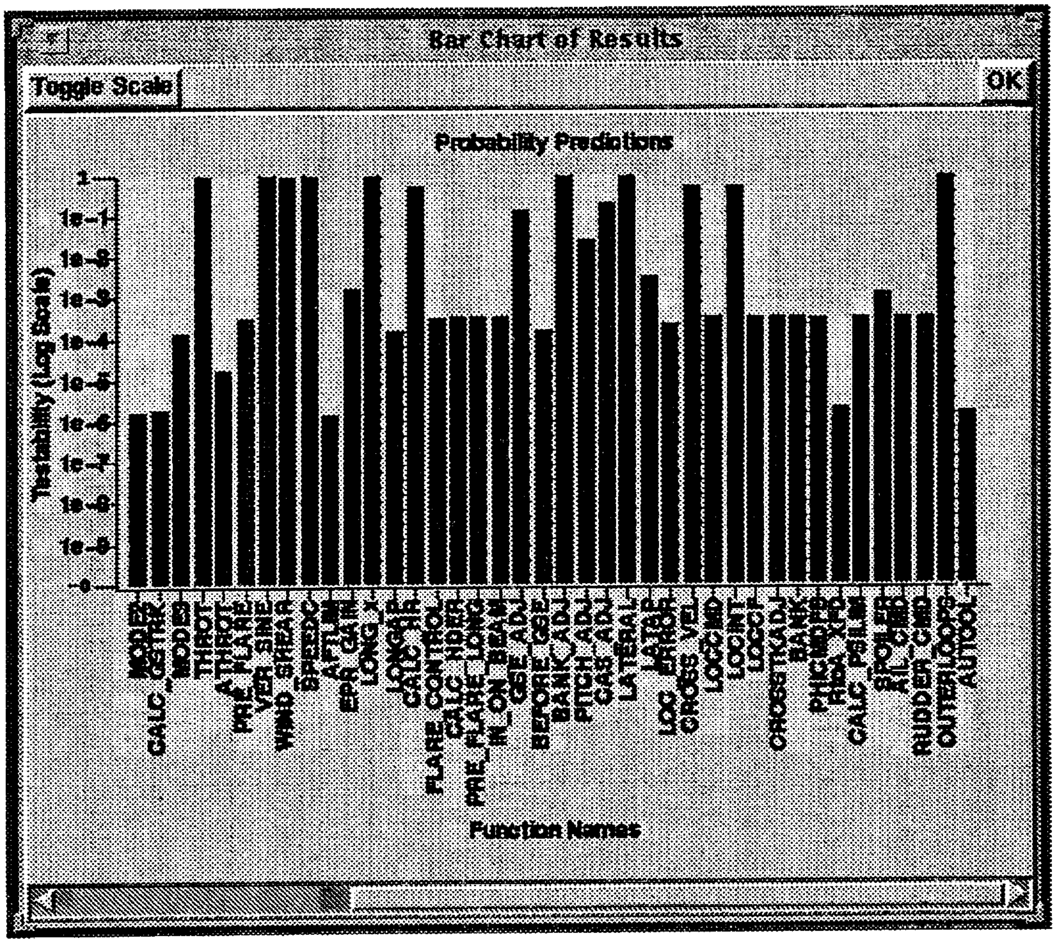
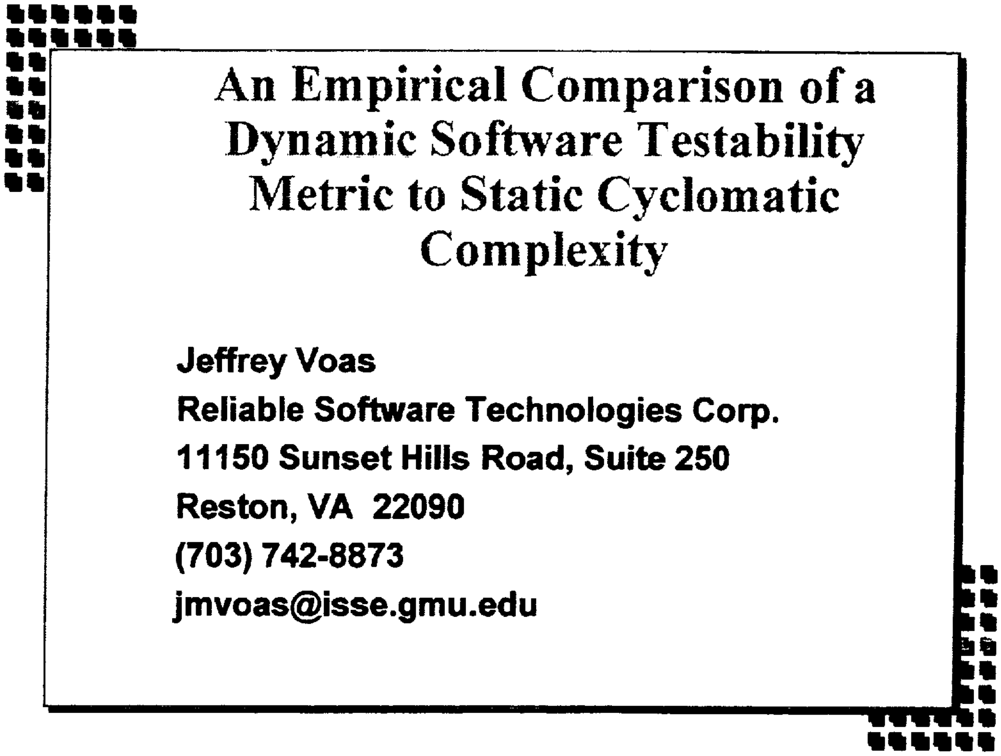
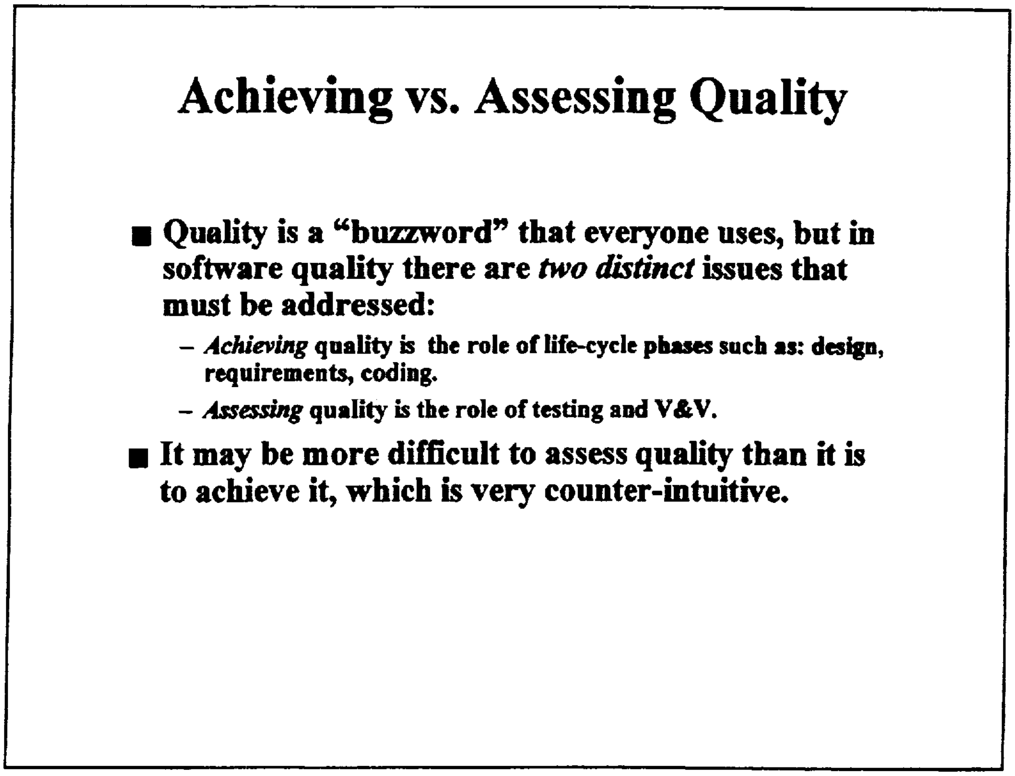


Figure 3: More function testabilities for autoland.



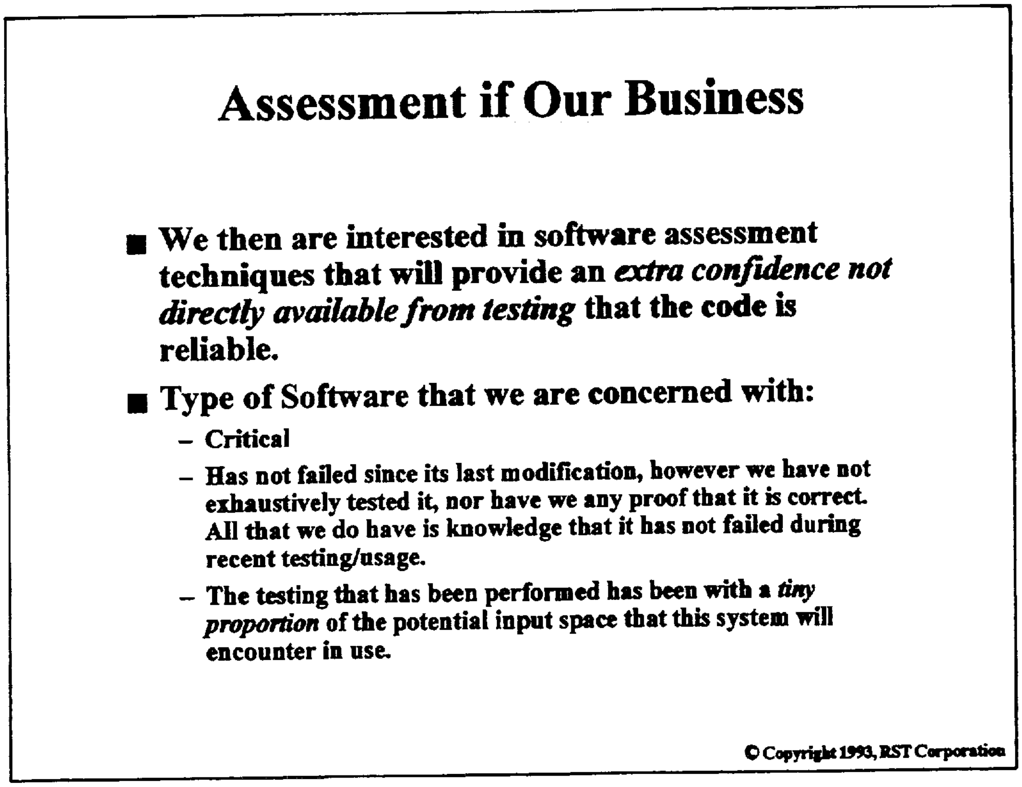
An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity

Jeffrey Voas
Reliable Software Technologies Corp.
11150 Sunset Hills Road, Suite 250
Reston, VA 22090
(703) 742-8873
jmvoas@isse.gmu.edu



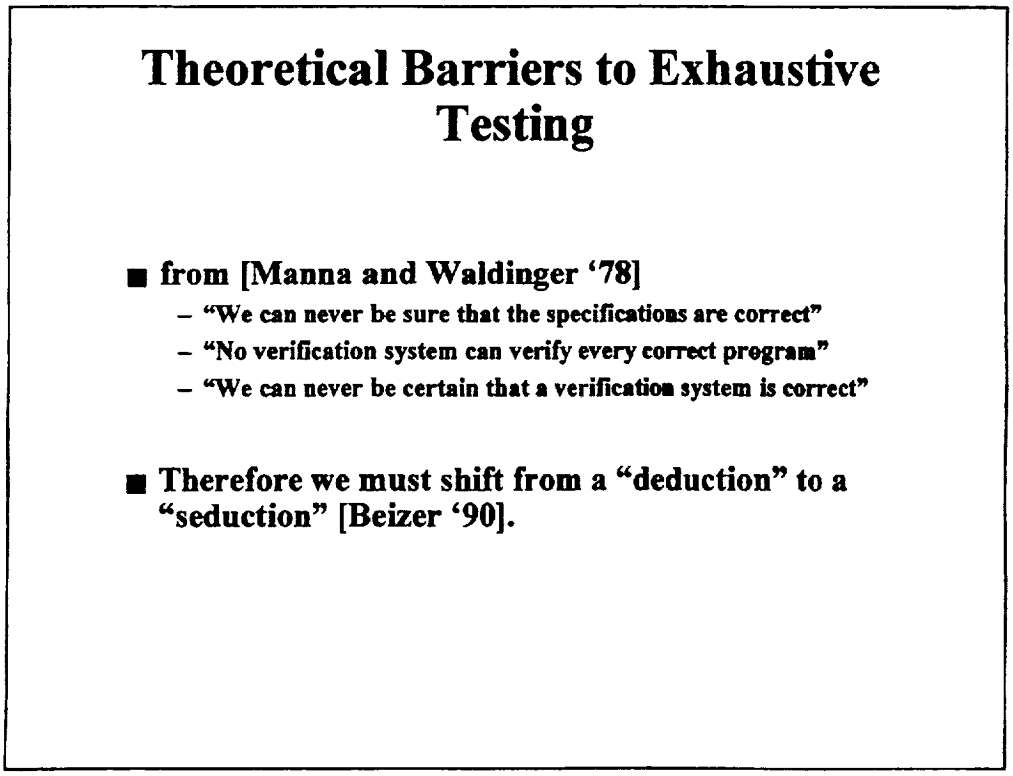
Achieving vs. Assessing Quality

- **Quality is a "buzzword" that everyone uses, but in software quality there are *two distinct* issues that must be addressed:**
  - ***Achieving* quality is the role of life-cycle phases such as: design, requirements, coding.**
  - ***Assessing* quality is the role of testing and V&V.**
- **It may be more difficult to assess quality than it is to achieve it, which is very counter-intuitive.**



Assessment if Our Business

- We then are interested in software assessment techniques that will provide an *extra confidence not directly available from testing* that the code is reliable.
- Type of Software that we are concerned with:
  - Critical
  - Has not failed since its last modification, however we have not exhaustively tested it, nor have we any proof that it is correct. All that we do have is knowledge that it has not failed during recent testing/usage.
  - The testing that has been performed has been with a *tiny proportion* of the potential input space that this system will encounter in use.



Theoretical Barriers to Exhaustive Testing

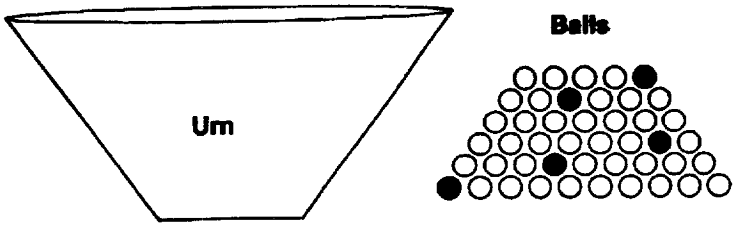
- from [Manna and Waldinger '78]
  - "We can never be sure that the specifications are correct"
  - "No verification system can verify every correct program"
  - "We can never be certain that a verification system is correct"
- Therefore we must shift from a "deduction" to a "seduction" [Beizer '90].

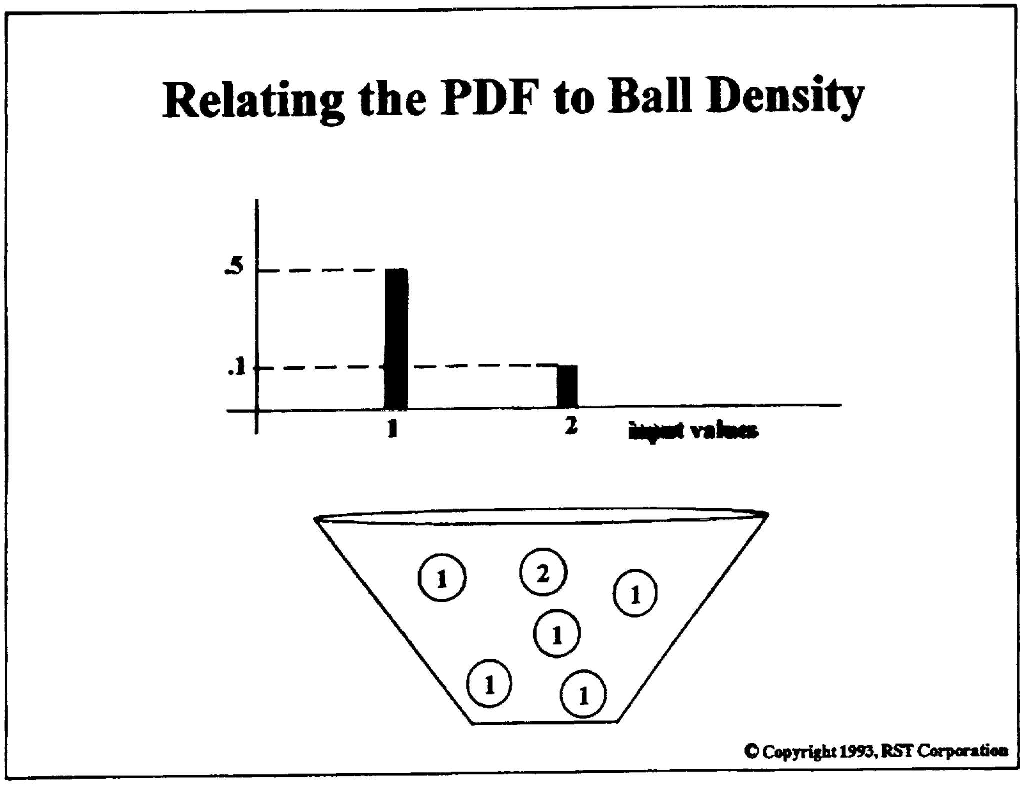
Difference between Testing and Testability

- Testing is defined with respect to some "authority" that asserts whether an output is correct.
- Testability says nothing about correctness, but rather thc likelihood of incorrect output occurring.
- This is a fundamental difference that needs to be understood.

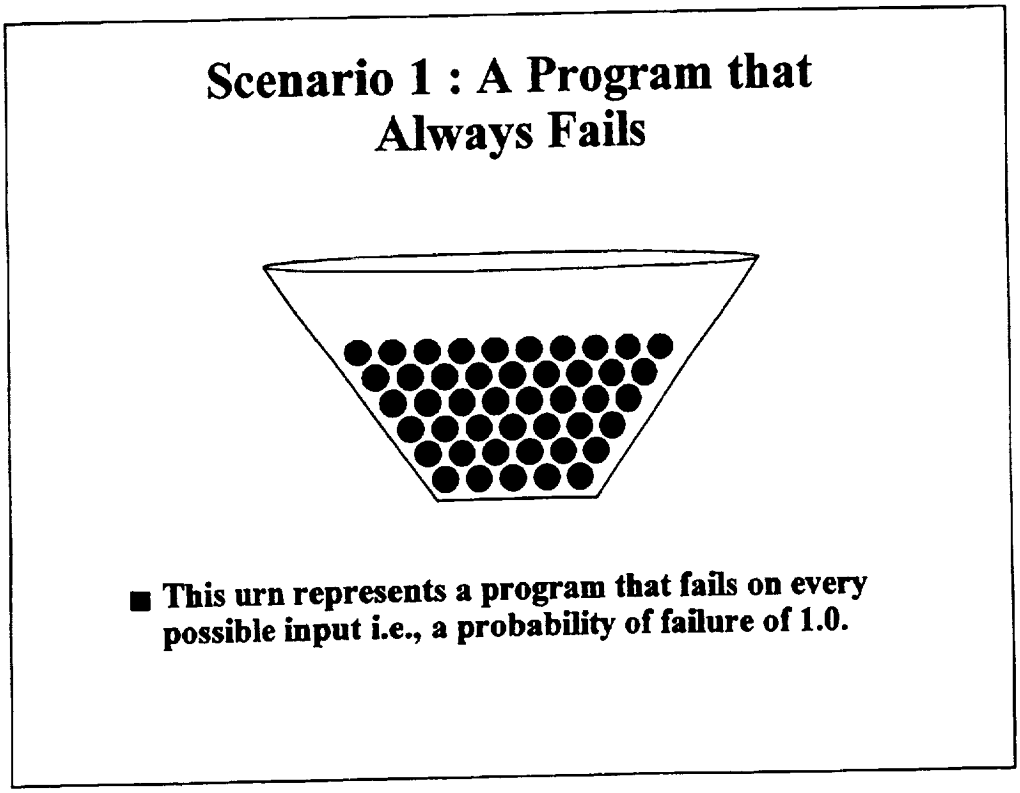
Balls and Urn

- Testing can be viewed as selecting different colored balls from an urn where:
  - Black ball = input on which program fails.
  - White ball = input on which program succeeds.
- Only when testing is exhaustive is there an "empty" urn.





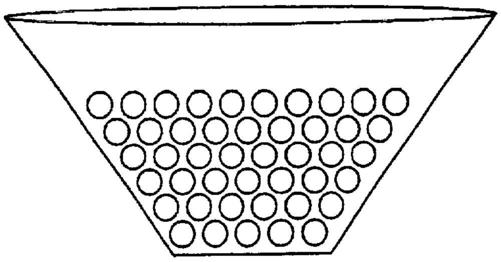
Relating the PDF to Ball Density



Scenario 1 : A Program that Always Fails

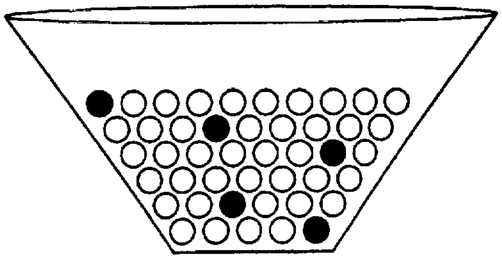
- This urn represents a program that fails on every possible input i.e., a probability of failure of 1.0.

Scenario 2 : A Correct Program



- This urn represents a program that succeeds on every possible input i.e., a probability of failure of 0.0.

Scenario 3: A Typical Program



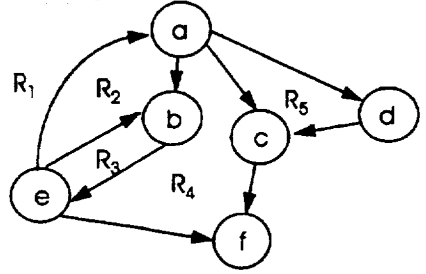
- This urn represents virtually all software in use today.

Traditional Definition of Software Testability

- **Definition: the ability of a system to be easily and thoroughly tested, where thoroughly means that a particular coverage metric is achieved (e.g. statement coverage, branch coverage).**
- **Based on the weak assumption that covering code means no faults remaining.**
- **Example: McCabe's Cyclomatic Complexity Metric [*IEEE Transactions on Software Engineering*, 1976].**

Cyclomatic complexity

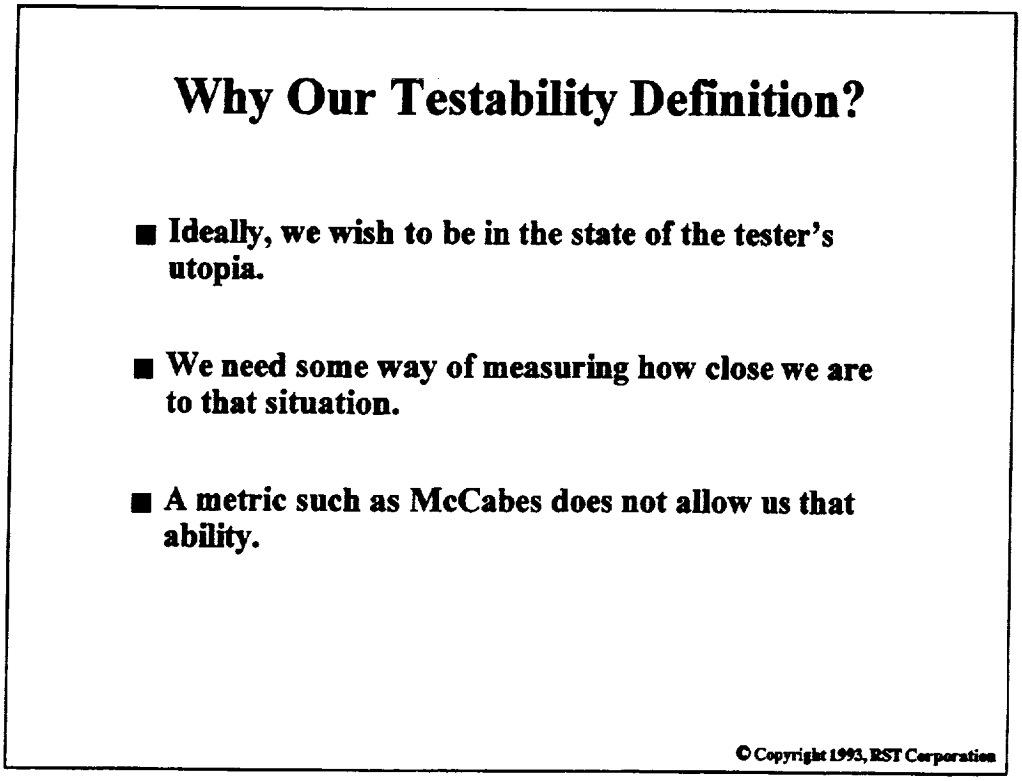
V(G) = # of regions
or
V(G) = L - N + 2P
where:
L = # of links
N = # of nodes
P = disconnected parts



V(G) = 5

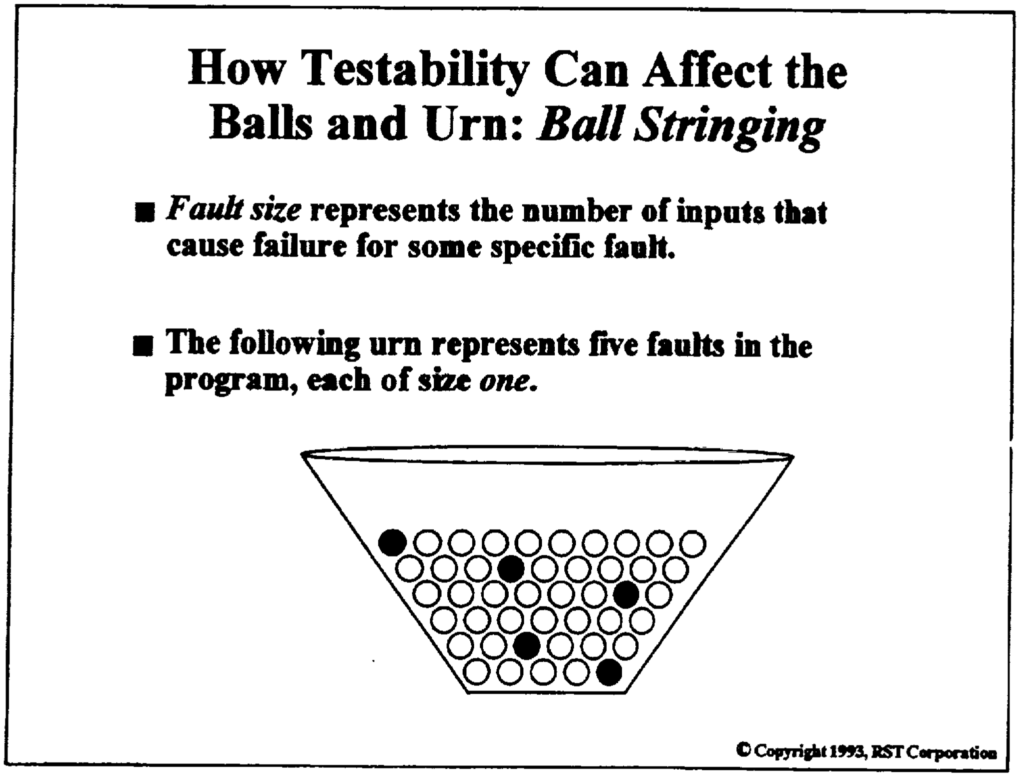
Our Definition of Software Testability

- **Definition: a *prediction* of the probability that existing faults will be revealed during testing according to some testing scheme *D*. (*D* is some input representation)**
- **If there is a fault at a particular location, how likely is it you will see the fault as a failure during testing according to *D*.**
- **If faults are unlikely to cause failures then it is obvious that the fault will be difficult to detect during testing.**



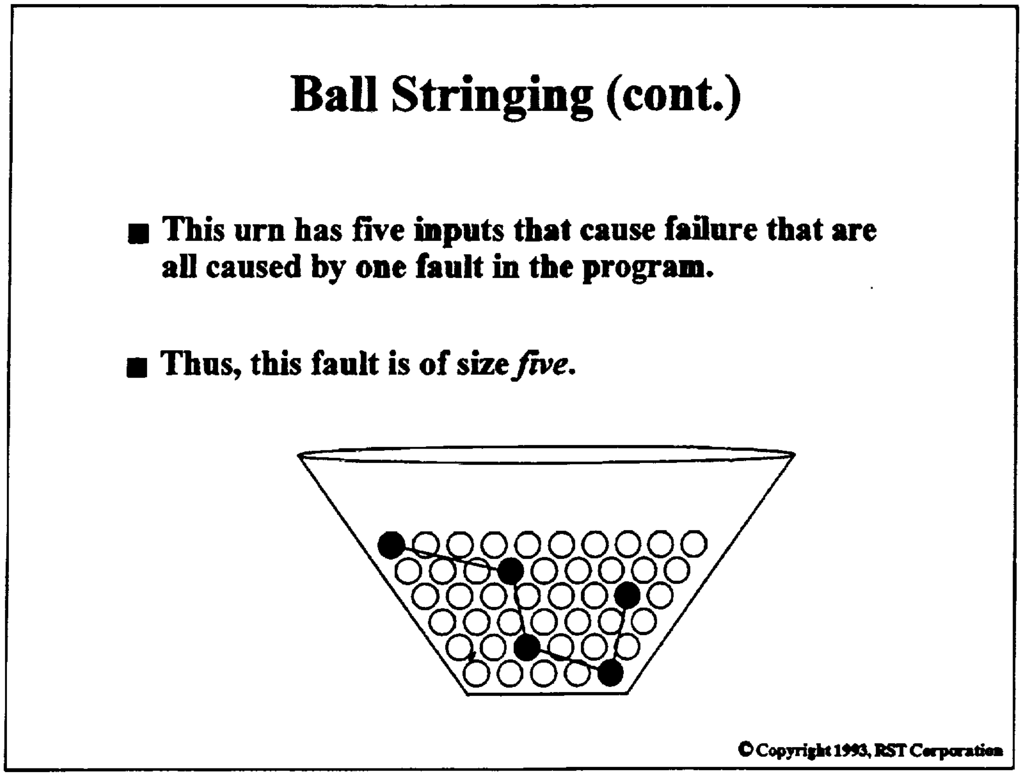
Why Our Testability Definition?

- Ideally, we wish to be in the state of the tester's utopia.
- We need some way of measuring how close we are to that situation.
- A metric such as McCabes does not allow us that ability.



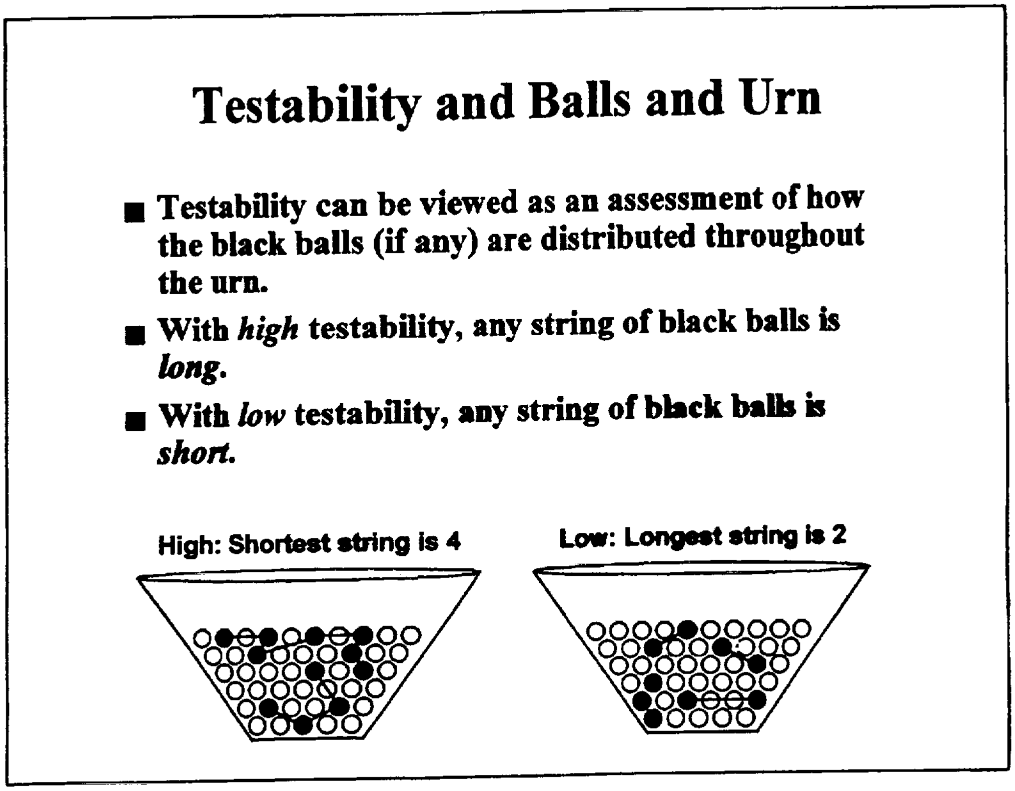
How Testability Can Affect the Balls and Urn: *Ball Stringing*

- *Fault size* represents the number of inputs that cause failure for some specific fault.
- The following urn represents five faults in the program, each of size *one.*



Ball Stringing (cont.)

- This urn has five inputs that cause failure that are all caused by one fault in the program.
- Thus, this fault is of size *five*.



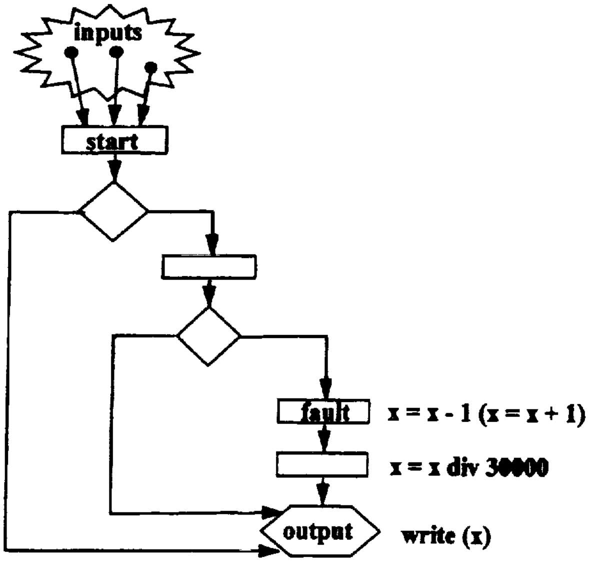
Testability and Balls and Urn

- Testability can be viewed as an assessment of how the black balls (if any) are distributed throughout the urn.
- With ***high*** testability, any string of black balls is ***long***.
- With *low* testability, any string of black balls is ***short***.

High: Shortest string is 4

Low: Longest string is 2

Example of Hiding Fault



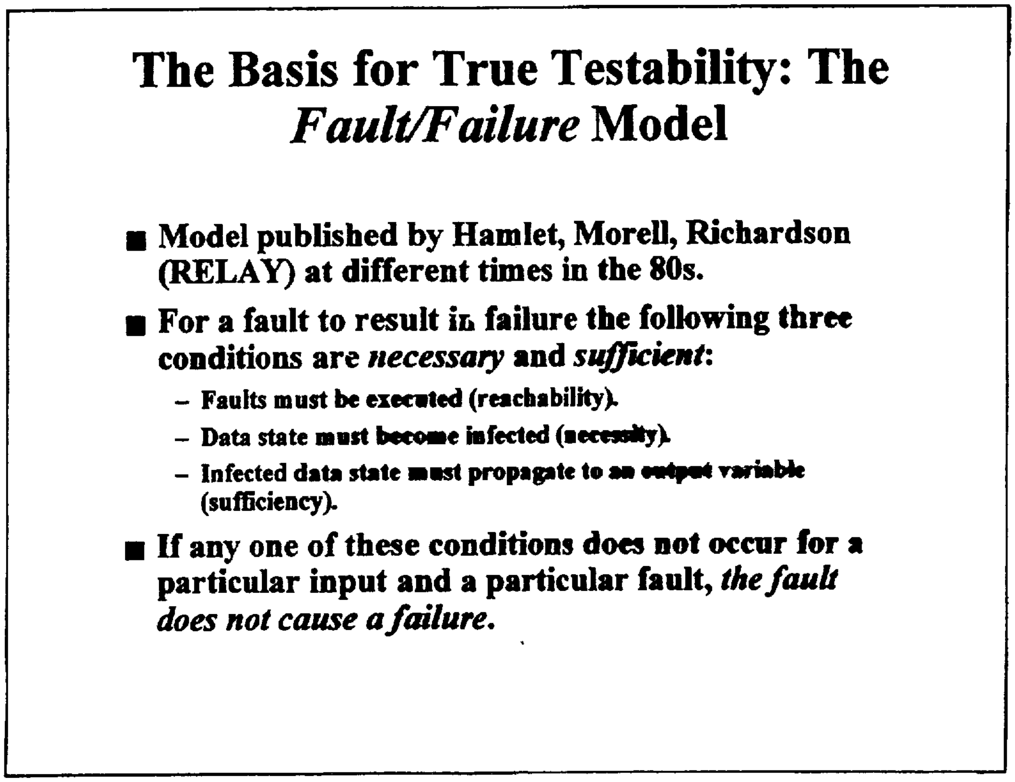
x = x - 1 (x = x + 1)

x = x div 30000

write (x)

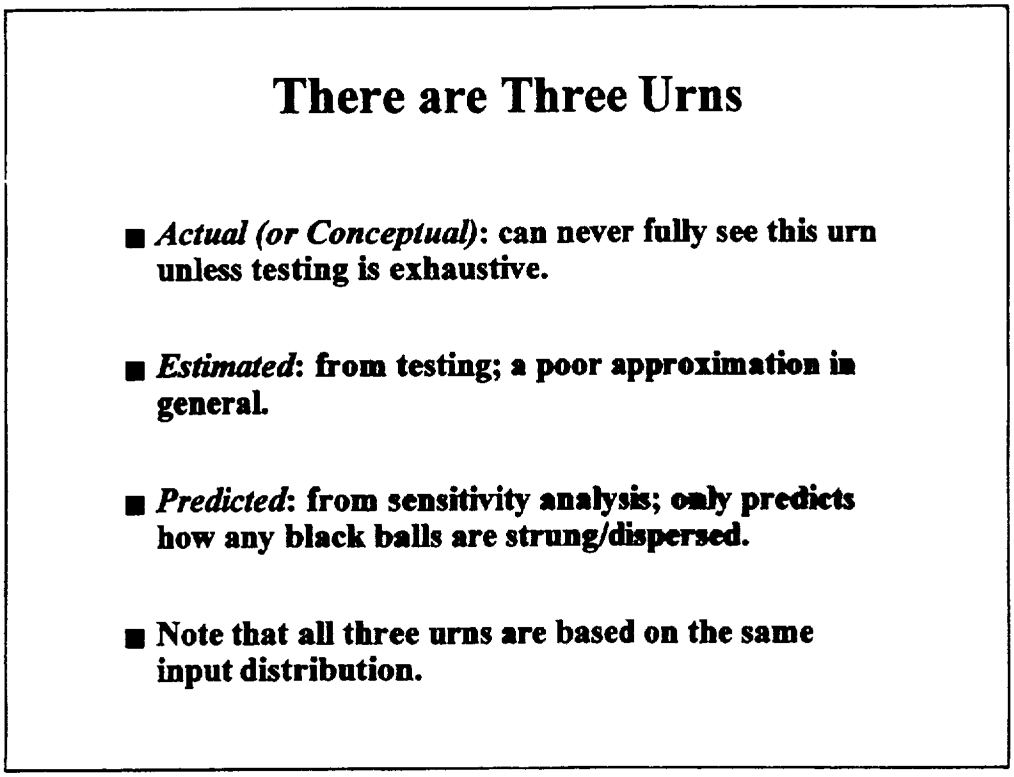
How Will We Predict the "Coloring" and "Stringing" Within the Urn?

- Use lower level of code abstraction than McCabe.
- Use mutation analysis techniques.
- Approximate all 3 conditions of the fault/failure model.
- Increase error classes considered beyond those generally considered by fault-based/error-based techniques.
- Use dynamic code analysis instead of static code analysis.



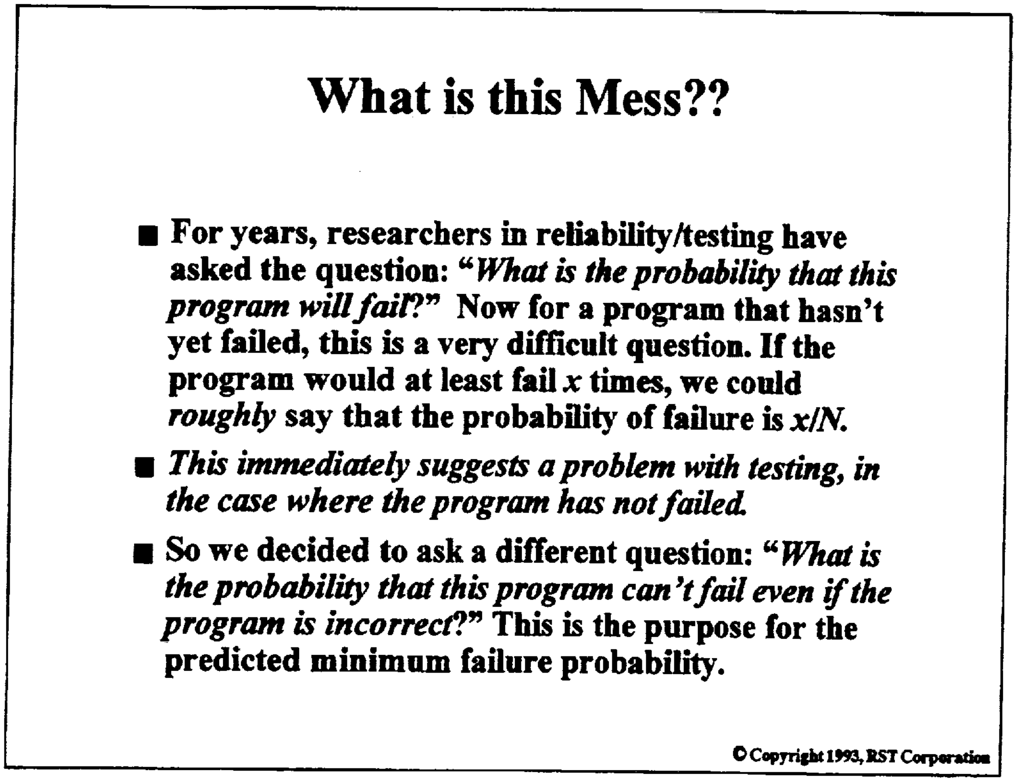
The Basis for True Testability: The *Fault/Failure* Model

- **Model published by Hamlet, Morell, Richardson (RELAY) at different times in the 80s.**
- **For a fault to result in failure the following three conditions are *necessary* and *sufficient*:**
  - **Faults must be executed (reachability).**
  - **Data state must become infected (necessity).**
  - **Infected data state must propagate to an output variable (sufficiency).**
- **If any one of these conditions does not occur for a particular input and a particular fault, *the fault does not cause a failure.***



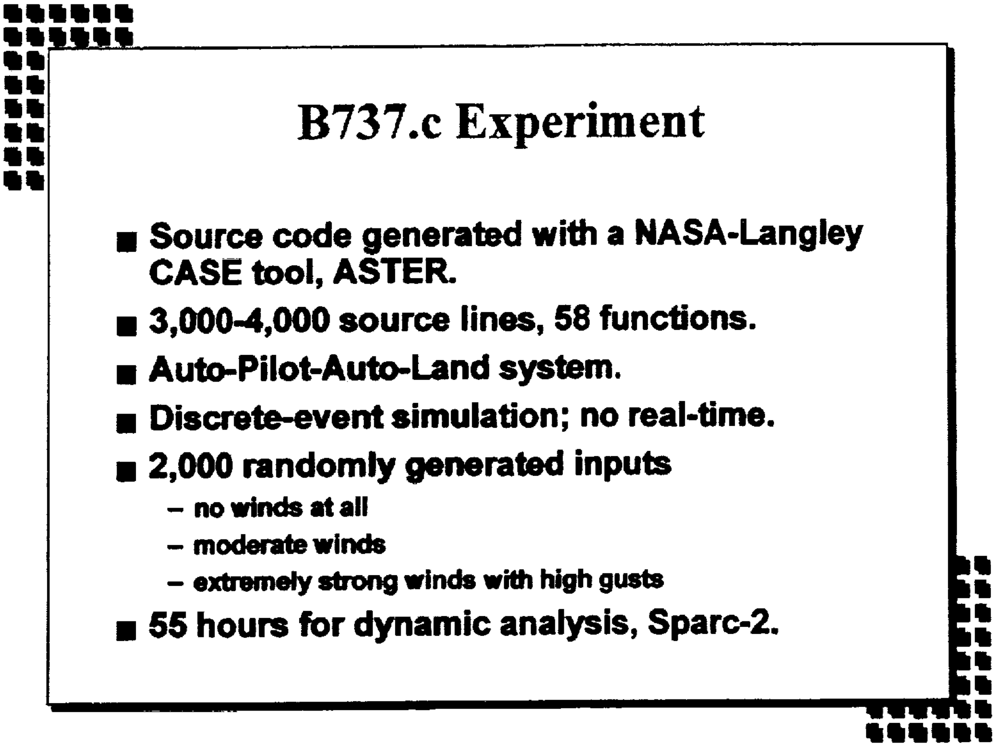
There are Three Urns

- ***Actual (or Conceptual)*: can never fully see this urn unless testing is exhaustive.**
- ***Estimated*: from testing; a poor approximation in general.**
- ***Predicted*: from sensitivity analysis; only predicts how any black balls are strung/dispersed.**
- **Note that all three urns are based on the same input distribution.**



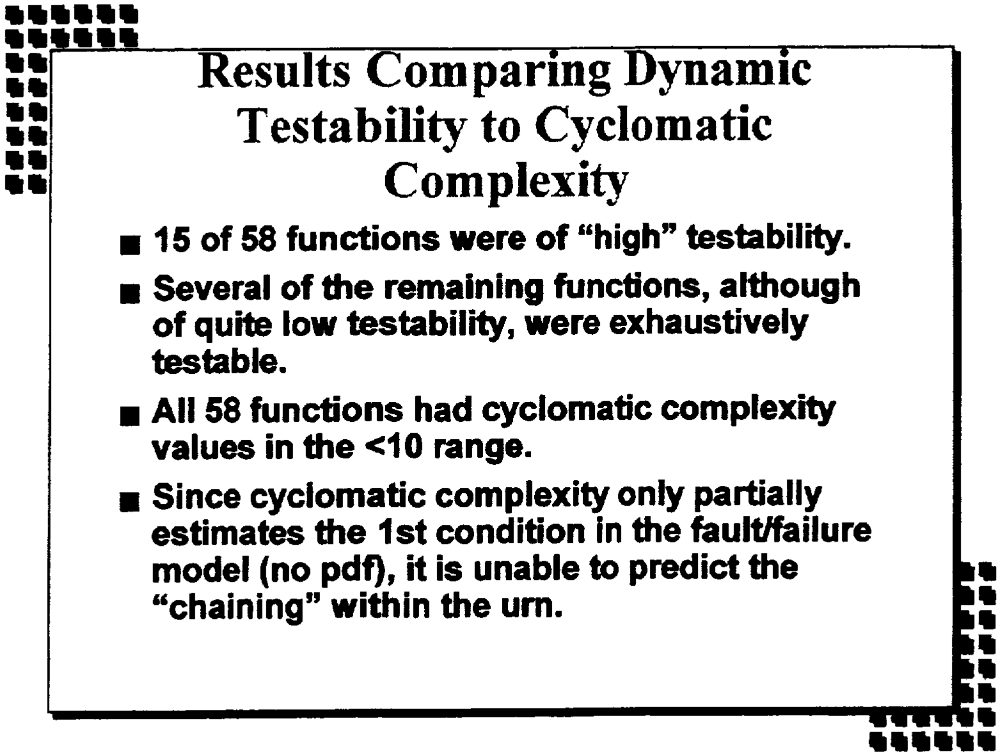
What is this Mess??

- For years, researchers in reliability/testing have asked the question: "*What is the probability that this program will fail?*" Now for a program that hasn't yet failed, this is a very difficult question. If the program would at least fail *x* times, we could *roughly* say that the probability of failure is *x/N*.
- *This immediately suggests a problem with testing, in the case where the program has not failed.*
- So we decided to ask a different question: "*What is the probability that this program can't fail even if the program is incorrect?*" This is the purpose for the predicted minimum failure probability.



B737.c Experiment

- Source code generated with a NASA-Langley CASE tool, ASTER.
- 3,000-4,000 source lines, 58 functions.
- Auto-Pilot-Auto-Land system.
- Discrete-event simulation; no real-time.
- 2,000 randomly generated inputs
  - no winds at all
  - moderate winds
  - extremely strong winds with high gusts
- 55 hours for dynamic analysis, Sparc-2.



Results Comparing Dynamic Testability to Cyclomatic Complexity

- **15 of 58 functions were of "high" testability.**
- **Several of the remaining functions, although of quite low testability, were exhaustively testable.**
- **All 58 functions had cyclomatic complexity values in the <10 range.**
- **Since cyclomatic complexity only partially estimates the 1st condition in the fault/failure model (no pdf), it is unable to predict the "chaining" within the urn.**