Testing, Testing, Testing.. an ACS Survey of Industry Practice

By Karl Reed, FACS,HLM, MSc,ARMIT
Director, Computer Systems and Software Engineering Board
Visiting Professor
School of Information Technology[1]
Bond University

A CONFLICT OF INTEREST!

Dear reader, firstly, I teach about testing and research in the area. I have a conflict of interest, even beyond this. I am part of a research project, funded by ACS through its SERCC, to undertake the Australian end of a regional survey of industry practice… The purpose of this piece is, dear reader, to.. well, .. get you in!

1. To Test or Not to Test.. That is the Question

I am sitting in the gate lounge[2] of an unnamed international airport, and there is an announcement. Yes, there are always announcements, but this one is different.. "Crash and Burn Airlines", she says, "are offering you a choice of aircraft for your flight to Singapore today. The XX 380 on the left, like the YY 370 on the right, has computer controlled fly-by-wire flight control systems. The '380's systems have been thoroughly tested, and the 370's have been formally verified and proven correct, but, they have NOT been tested, and this is the aircraft's very first flight! The YY company believes its OK.. and, today, dear customers, you can chose which one you will fly on."

If you think this is ridiculous, then you may have missed some of the most important philosophical and practical arguments in our field. The problem looks something like this. Pick any modest-sized program with a realistic input specification, and try to construct a set of tests that will guarantee that the program meets the specification. For even simple programs, many hundreds of test-cases will be produced by any test-generation method you care to choose. But, if the program passes all your tests, will this mean it is "correct"?

The answer is .. "you can't be sure". We will discuss the reasons for this shortly. However, if you find an error (or fault), then its possible to say something definite… "We found a fault (and hopefully corrected it)[3]". This has lead to the oft quoted aphorism, due to Edgar Dijkstra, "Testing cannot prove the absence of errors" By implication, it can only prove their presence. This follows, it is argued, because test-sets are limited in number (finite) while the set of all possible inputs for any real program could extremely large (near infinite, consider a compiler). You can add to this some serious doubts about the "true" (read "theoretical") usefulness of test generation techniques (more later), and the argument for arguing against testing can seem to have more than academic validity!

If, as we shall see, testing is hard, and (as we shall see) unable to locate all the errors which may exist in code, then why persist at all? Firstly, despite all the problems, thorough testing DOES locate faults, whose removal increases a systems quality. Secondly, testing need not be limited to simply the detection of faults, it should also address usability issues. Thirdly, the test-sets that a system passes, if properly constructed, can define an operational envelope. It would be possible to restrict the

---

[1] On leave from the Department of Computer Science and Computer Engineering, La Trobe University, Bundoora 3083, Vic. Australia
[2] This is not original. I forget who first made this point, but is told repeatedly..
[3] Of course there are four options here…One, fix the fault, Two add it the list of known bugs, Three hope the users won't notice,

systems operation to this "envelope" of valid operations, even if they were a limited set of the intended functionality. In mission-critical or safety-critical applications, this is essential. In more mundane applications, it would certainly improve users effectiveness and save them vast amounts of inconvenience. In addition, it is consistent with (rather, it inverts) concepts of incremental delivery advocated by some methodologies (See for example Pressman **[Pres2001]** p.35)

At this point, we need to draw attention to a quite old area of research in computer science called "formal methods". In practice, this is also known as "program proving", and the idea arose quite early in our field. The basic idea is that, since a program is a series of transforms of data, then, if we can describe these transforms mathematically in an appropriate system, then we should be able to regard each program as a theorum which can be proved correct or incorrect. This field can be said to have come of age when C.A.R. Hoare **[Hoa1969]** published a series of proofs of algorithms and analyses of problems that demonstrated that programs could indeed be "proved". Hoare built on earlier work by Floyd. (The paper **[Elspa1972]** contains an excellent survey area. Chapter 25 of Pressman contains a useful overview as well **[Press2001]**). A few years later, Cliff Jones at the IBM labs in Vienna led an effort which produced the Vienna Definition Method and Language.(See his much latter text book for an excellent presentation **[Jone1990]**)

Program proving has come a long way in the last 35 years. There have been a large number of formal systems developed, and quite extensive experience in their use. There have been significant claims of success. In addition, the approach is now often applied to specifications of programs, where it can be used to prove that they are correct, some thing generally easier than proving the program itself. From a teaching perspective, one interesting example is that of Hall's. This is based upon the real-life case where the company Praxis used the formal system Z to specify an air-traffic control system. Hall presented this as a tutorial at the 1994 ICSE in Sorrento. The general flavour of the work can be seen from **[Hall2002].**

But, program proving suffers from the same problems that programming itself does. Firstly, the proofs, like any mathematical proof, can contain errors (there are famous examples of published program proofs containing errors. Secondly, the methods require special training and skill, and some mathematical aptitude[4].. and hence are not widely used. In addition, it is difficult to formally verify or prove code containing many of today's favourite programming devices. For example, the critical parts of the Boeing 777 digital flight control systems are reputed to be written in a variant of Ada which has no dynamic storage allocation, and no generic packages. An example of the kind of restrictions can be found in **[Carr1990]**. There has also been a significant amount of work on the use of formal specifications to derive test cases, what is called "specification based testing". This is a little different from the Model Checking mentioned later, and work by Phil Stocks and David Carrington provides a useful example **[Stock1996]**).

Where formal[5] methods are concerned, a major debate washes across the community. Claim and counter claim are made. Perhaps the best accessible discussion of this can be found by Norman Fenton and Shari Pfleeger **[Fent1997]** in Shari Pfleeger and Les Hatton's paper of 1997 **[Pfle1997].** The survey by Carre and Wing is more optimistic**[Carr1996].**

---

[4] Of course, this raises an interesting aside. Engineers generally are expected to be proficient in calculus and applied maths to fourth year university level. Why is it that IT community has such an abhorrence of maths ?

[5] We should add that the University of Queensland's Software Verification Research Centre is a world-class contributer in this area . See http://svrc.it.uq.edu.au/

A middle ground was proposed by Harlan Mills in the mid 1980's **[Ling1994]** with the Clean-Room approach. This required a (semi) formal approach to design, coupled with partial deliveries and statistical testing.. however, the developer/programmers are not allowed to unit test their code! Instead, a test group certified the code. Again the jury is out as to whether this is a practical way of building software.

But the debate on "why test software" was given a nasty twist in 1987 when Vic Basili (now a Visiting Professor at UNSW, and a collaborator of Ross Jeffery) published a study that showed that errors could be found in programs more effectively by reading them then testing them! Vic conducted three error locating experiments[6] **[Basil1987]** on a single piece of software as follows:-

A/ Functional-testing-The first groups were given only the spec and the executable, and allowed to execute the program, using equivalence class and boundary value analysis,

B/ Structural-testing- The second groups were given the source code and the spec, and allowed to execute the program with the goal of achieving 100% statement coverage.

C/ Code-Reading- The third groups were NOT given the executable code, but were given the source code and the spec, and could only read the program

The result was that the experienced cohort in group C/ was significantly more effective in finding errors in the code than their counterparts in the other two! This may not be surprising in terms of what we now know about inspections, however, it was not an outcome that was expected at the time.

As we speak, however, formal methods are still an area for specialists, so, we come back to testing…

2. What do we know about testing?

It turns out that we know quite a lot about testing. We have been doing if since the inception of the computer field. Some of the things that we know from research and practice, however, are rather disquieting. For example, it is known that in practice, it is difficult and unusual to achieve 100% statement coverage or path coverage no matter how carefully we generate input cases (**[Basi1987]**). By the way, this fact was known well before the problem of "code bloat" was raised by the practices of some large microprocessor s/w developers. A seminal paper[7] by Peter Herman **[Herm1976]**, published in the Australian computer Journal in 1976 described a testing tool based on data-flow anaylsis, that measured "coverages" and proposed special steps to exercise code that was not covered by normal means.

Over the years, a large number of testing techniques have evolved. Some have extremely "formal" bases, and some are less formal. Approaches are divided into "white-box" and "back-box", according to whether they involved detailed analysis of the code, or allow testers to work from the specification. Not all "back-box" tests deserve that title. Several (equivalence-class and boundary value) require the tester to "..partition the input into equivalence classes that have the property that any test chosen from such a class is equivalent to any other test in the class". This means only one test is needed instead of many. However, a little careful thought leads one to conclude that for the tests to be equivalent in this way, then they must all execute the same code!

---

[6] I am simplifying things. There in fact more than three experiments, split across experienced programmers and students.

[7] This paper is extraordinarily widely cited by serious researchers in data-flow testing

Inverting this, it becomes clear that if they don't execute the same code, then they are not equivalent! This suggests that the only way of being sure that the "partitions" are correct is by looking at the code!

Some of the earliest work in area (see Richardson **[Rich1985]** for example), indeed proposed that this was exactly how this (equivalence class) testing should be performed. Recent work by T. Y . and his co-workers also proposes that these two techniques "black box and white box" need to be used jointly **[Chen2000]** However, as often happens, the popularised version of the approach (see for example. Myers or Pressman) ignores the hard bit. There has been considerable work in attempting to automate the generation of test-cases based upon formalized specifications that are "parsable" in some sense. One important approach which treads this path is that of Model Checking**[Atle1993].** *"Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds for that model. Roughly speaking, the check is performed as an exhautsive state space seaarch that is gauranted to terminate since the model is finite*" **[Carr1996]** This approach is now supported by tools which are claimed to be effective in, well checking the models.

There seem to be two core elements of white box testing. The first is "coverage" based, and the second is fault location. By coverage, we mean that a set of tests are constructed, and their effectiveness gauged by stating that n% of some program feature is exercised by them.(Zhu **[Zhu1997]** is a reasonable summary of the coverage topic). For example, we could report that a particular test-set caused 90% of the statements to be exercised. Obvious candidates for "coverage" are:-

A/ Statements

B/ Paths,

C/ Procedures/Classes

D/ Procedure Calls/Class/Method invocations

F/ Predicates

G/ Exception or event raising

H/ Exception or event interception

The list of features worth including in coverage testing is language dependant. For example, a problem with languages such as Ada and the OO languages arise with procedures/classes that can be invoked with a wide variety of parameter types. In most older languages, it is possible to call a procedure ONLY with specific parameter types. All this has changed with the OO languages.

Again, it can be difficult attach meaning to some of these coverage measures, and the there has been considerable evolution in the "features" considered worth covering. An interesting extension of applying data-flow issues to testing has been the "def-use" coverage measures proposed by Weyuker and her co-workers **[Fran1988]**. The work builds upon the very vast (and by our field rather ancient) body of knowledge in optimising compilers, and is called "def[8]-use chaining". The "formal" approach to this rather is complex, but, the idea, as I encapsulate it for my students, is "*In general, all results computer in a program are intended to be used to produce a final outcome. Hence, if one can find a path through a program which produces a final outcome, AND there are unused variable assignments on that path, the there may be an error, and you need to check that this is intentional"*. Of course, we sometimes write

[8] A variable which has been assigned a value is said to have been "defined".

"opportunistic" code, in which we computer partial results with the intention of discarding them, but, we do need to be sure!

Achieving "all statement" coverage does not guarantee that all paths have been covered, and, if all paths have been covered, this will not guarantee that all unused "defs" will be found. The scale of the problem can be seen from a very readable paper by Weiser and company **[Weis1985]**. A tutorial on the problems of testing, this paper describes a number of difficulties of which the following are examples:-

The **sub-expression** problem is demonstrated by a statement of the kind:-

A:= B+(3-X)*(X-5). If we test this with inputs that cause X to either 3 or 5, the right sub-expression will always be zero, and an error in either of the factors is masked.

The **multi-value** problem could also be seen from this if, for the same statement, if B was always 6 for the two test-cases above, the value assigned to A is always 6.

Hence, code further down the execution path may not be properly exercised, since it sees only one value for A.

Adding both sub-expression and muli-value to our coverage list makes life even more interesting, and the Wieser et. al. Paper includes more.

. The use of OO creates additional problems for coverage and unit testing. Unit testing is complicated by the fact that OO systems often consist of relatively small classes whose functionality is inherited from elsewhere, so the true function of the class only "exists" when it is executed in the appropriate hierarchy. Pressman devotes a complete chapter to this subject (chap 23, **[Pres2001]**) and advocates that the testing of OO systems should really begin by a detailed examination of the analysis and design models, to ascertain the semantic context of classes under test. We have already mentioned the problems created by the "genericity" of parameters in the absence of strong typing (or rather, in the presence of type overloading). In practice, a class/method (generic package, etc.) cannot be considered to have been tested unless it is exercised with all the possible parameter types for which operations within it are valid. It is worth pointing out the Paul Strooper at University of Queensland, has developed a special purpose test harnesses to support OO testing**[Hoff1998]**.

The "fault-location" arm of white-box testing is related, in my view, to debugging and to "test-quality". Once a fault or error (i.e. a deviation from the spec.) has been identified, the cause must be found in the code. While of considerable interest, we do not have space here to discuss more than two examples of this approach. In 1970, Ernie Zimmer, a colleague working with an Australian team at Ericsson's in Stockholm, developed a test harnesses[9] for the AKE130 family of computer controlled telephone exchanges. The test-harness relied upon the fact the names of data-items, system and call states, device types, switches and their controlling features, and operations, as described in specifications, had almost completely one-to-one mappings onto the code. In other words, if a particular device has a register, and that register had some setting indicating that (say) a particular signal had been received, then all of these entities would have (documented) symbolic names in the code. Ernie developed a language for describing the expected behaviour of a system at the system-level, in terms of the values expected in a series of data items, at a sequence of points traversed during execution. The test-harness would check that indeed the values and points reached were correct, and could re-set incorrect values to allow execution to continue. Test harnesses do not seem to be as popular as they once were.

---

[9] The are other examples of test harnesses.

While I am not sure that the GUARD system was derived from the test-harness concept directly, it can certainly be considered to be a generalisation. Developed by Abramson and Sosic **[Sosi97]** the system uses the data generated by a version known to be a valid, working version of a program, and allows it to be compared with the same data generated by a modified version, or one which was ported to another platform. The interesting thing about GUARD is that it actually works by allowing both programs to be run 'simultaneously', either on one machine, or on two, and it may need to communicate over a network to make the comparisons. This has great value. GUARD is currently used for testing evolving computationally intensive programs, however, it is capable of being used to test, and debug, distributed systems.

Test-quality relates to having confidence that our tests would indeed find errors with they were there. The "white-box" aspect of this deals with the process of seeding of adding bugs to the code, and running the tests to see if they produce a detectable fault. Again there has been significant research on this. Knight looks at the capiry of tests to dect syntactic errors, **[Knig1985]** and Shooman explains how a statistical device called the "fish pond test" can be used to estimate the number of errors in a program[10] **[Shoo1983]**). The most extreme approach is known as "code-based mutation testing" **[King1991]** (to differentiate it from the more recent approach of "specification-based mutation testing", a black-box approach being investigated by my student, Tafline Murnane **[Murn01]**. In this, every single statement in a program is "mutated", one at a time, and tests run for each single mutation to see whether the output differs from an un-mutated version of the program. Each statement is mutated many times (generating many runs), since each syntactic element of the statement must be replaced by another (syntactically valid) element, until a mutant has been generated for all (of the syntactically valid) elements known to the program. This approach is not genuinely useful, since it generates large numbers of mutants and hence test-runs. However, it requires truly virtuostic skills in range of technologies to produce even vaguely usable systems[11].

4. Operational Testing

Earlier, we mentioned the possibility of using a set of test results to define an operational envelope for a system. We advocated constraining the system to that envelope via a filter. The result would be, we hope, a system containing (working) features that a customer wanted. However, it is also possible to invert this argument (N.B. I am not arguing that this was how Musa developed his operational approach), and to seek to test those functions which are used most often. John Musa**[Musa1993]** proposed that one should collect data on the actual usage patterns of a systems (or, develop them for a new system), and use them to construct tests which verified that the functionality required to support this actually worked. To achieve this, one collects statistical data on functional usage, and tests the most those most likely to be used first. In fact, Musa'a system seem to suggest that one should ONLY test for those case that appear in a usage profile. Since it is possible to deal with low-probability but critical functions by weighting them accordingly, the objection that these may be overlooked is dealt with. The approach is not dissimilar to that proposed for statistical testing in the Cleanroom methodology, where test-sequences are generated randomly as sequences of input events based upon frequency of occurrence of their constituents.

5. When to Stop Testing

---

[10] As with many of these approaches, it is hard to find useful experimental evidence that this is effective.

[11] Readers, as a "thought" exercise, might like to design a support system for this approach, without cheating and looking at the refernce.

When should testing stop is a question fundamental of practical and theoretical importance. In practice, we would like to know that either we have found all the faults in a systems, or, that it is no-longer economic to continue trying to find (what we hope will be minor) faults. To find all the faults in a system when we don't how many there were when it was delivered for testing, requires a means of estimating the number of faults based the results of the early phases of testing. We may also be aided by historical data which recorded the fault rates for particular teams producing similar software. The practical importance lies in a producer's need to control costs, pure and simple. The theoretical importance lies in solving a complex statistical problem which has great practical importance. There are a wide range of mathematical models, some rather intractable, which can be used to estimate the number of faults to be found (the "fish pond method has already been mentioned). Musa's**[Musa1990]** two models, the "basic" and the "logarithmic", provide tractable estimates, although I have not seen much recent work validating them. Ehrlich and company**[Ehrl1993]** present a rather theoretical analysis, whose methods are not easy for practioners to use, but which illustrate the mathematical sophistication involved.

From a pragmatic point of view, one can suggest a cyclic process which should be followed as the number of faults detected (cumulatively) begins to plateau as testing progresses. Check the test sets carefully, to ensure that they seem to have been thoroughly constructed. In particular, check that any prescriptive methods are being followed, and that the test-cases are reviewed to ensure that obvious[12] erroneous input has been included. If necessary, introduce new test-cases, and test some more. Else stop. If one has historic data, or can use one of the techniques for estimating the number of errors, then this should be done as well.

Implicit in this discussion is the possibility that a company may simply decide to release some software even though there are known errors. The purists amongst us would argue against this on the grounds that it is not good software engineering. Sadly, my daily experience with the software I depend on in my work place, Mac OS9.2, Navigator 4.7, Acrobat 4 and 5, and three generations of Microsoft products) clearly show that some suppliers consider their buggy products "good enough[13]". Currently, there is (yet another) debate in which senior figures in the Software Engineering community have become proponents of the "good enough" software movement. Each company, in the current IT climate, is making is own judgments here, balancing legal liabilities against product development cost against reputation and market penetration. The inevitable result will be that products will contain known errors.

We have already referred to the possibility of creating an "operational envelope" , a filter, which restricts the product to those inputs which it is known to process correctly……

Ultimately, being well informed on the art of testing, being aware of latest developments, and above all, understanding that no matter how matter what, reducing the inherent error rates in the programming process as at least as important as simply testing. As Whitaker points out in his excellent summary of testing **[Whit01ref]**, customers seem to report bugs, no matter how hard you test. Which suggests its much harder than some of us think!

6. The regional Survey-the Australian Connection. Australia, you're testing in it!

---

[12] It is often the case that, even when using prescriptive methods, this is not easy to do.

[13] As a caveat, one needs to state that despite this, the achievements of the suppliers are considerable, and the software impressive. It is possible that even the fault densities are also impressive. It is just that they should be even better!

In a short article such as this, it is not possible to cover all the issues associated with testing. Apart from the various types of testing that are possible, there is the question of exactly how those in industry actually perform testing, and, how effective is it? Anecdotal evidence is that more and more effort is being put into testing. How much more? Well, the purpose of this article is to encourage you to participate in a survey of software testing, so, the objective is to find out. The survey is being conducted by Swinburne University of Technology and La Trobe University, and is being financially supported by the ACS.

We do know that in recent years, software testing has been receiving increasing attention by the Australian IT industry. From the large number of delegates attending the first AsiaSTAR Conference held in Sydney last July and the popular demand of software testing training courses, there is little doubt that software testing has emerged as a specialised profession in the contemporary IT industry.

The proposed study will be based on a survey using a questionnaire intended to measure several key aspects of software testing practices in the Australian IT industry, such as the extent to which software testing practices have been adopted by organizations in Australia, and the subsequent benefits and related problems that have followed as a result of adopting these practices. The aim is to provide a clear indication of future trends and demands in the following areas of software testing:

- Industry practice
- Education and training
- Research.

Parallel research, using the same methodology, is being conducted by collaborators in South East Asian countries. Since the focus is wholly on industrial practice, one outcome of the research should be the identification of best practice that may enable the Australian software industry to develop competitive advantage.

The questions are designed to gather information from software testing practitioners and management in the areas of software testing techniques, automated tools, training and education, standards and external consultancy. Hypotheses will be tested against the collected data to derive measures and trends for the current and future software testing industry in Australia. For the survey to be effective, we need responses from a large number of software developers, whether they are heavily engaged in testing or not. The survey team is..

Doug. Grant, Swinburne Univ. Tech, Principal Investigator,
T. Y. Chen, and S. Ng, Swinburne Univ. Tech, Principal Investigator
T. Murnane and K. Reed, La Trobe University.

Companies interested in participating should contact the Survey Coordinator,

Dr. Sebastian Ng.
School of Information Technology
Swinburne University of Technology
John St.
HAWTHORN, VIC
Email:- sng@it.swin.edu.au
Ph:- +61 (0) 3 9214 8666

7. Conclusion and Acknowledgements

It has not been possible to deal with important current issues such as web-application and script-based testing in this article. The references have chosen for their relevance and accessibility, however, they may not always be the best on the topic concerned. The author gratefully acknowledges input and assistance from Phil Stokes (Bond University) and Tafline Murnane (La Trobe) who proofed the drafts, and provided some key references, and Dave Abramson (Monash), and Paul Strooper (University of Queensland). In the end, however, any errors of fact and sins of omission are the responsibility of the author.

8. Refs

[Atle1993] Atlee,J.M. and Gannon, M *State-Based Model Checking of Event-Driven System Requirements* IEEE trans. On Software Engineering, Vol 19 No 1, January 1993 pp24-40

[Basi1984] Basili, V. R. and Ramsey, G.. *Structural coverage of functional testing*. Tech. Rep. TR- 1442, Department of Computer Science, University of Maryland at College Park, Sept. 1984

[Basi1987] Victor R. Basili and Richard Selby, *Comparing the Effectiveness of Software Testing Strategies*, IEEE Transactions on Software Engineering, pp 1278-1296, December 1987.

[Carr1990] Bernard Carré , Jonathan Garnsworthy *SPARK—an annotated Ada subset for safety-critical programming* Proceedings of the conference on TRI-ADA '90 December 1990

[Chen2000] Chen, T.Y, Tang, S.F, Poon, P. L. and Yu, Y. T *White on Black: A White-Box Approach to Selectinbg Black-Box-Generated Test Cases* Proceedings of the First Asian Pacific Conference on Quality Software, IEEE Computer Society, Hong Kong 2000

[Clar1996] Edmund M. Clarke , Jeannette M. Wing Formal Mathods : State of the Art and Future Directions ACM Computing Surveys December 1996 Volume 28 Issue 4

[Ehrl1993] W Ehrlich, B Prasanna, J Stampfel, J Wu, "Determining the Cost of Stop-Test Decision", *IEEE Software, March 1993, Pages 33-42.*

[Elsp1972] Elspas, B., Levitt, K. N., Waldinger, R. J. and Waksman, A *An Assessment of Techniques for Proving Program Correctness* ACM Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147

[Fent1997] Fenton,N and Pfleeger, S.L. *Do Formal methods Always Deliver?* Side-bar in [Pfle1997]

[Fran1988] P G Frankl, E J Weyuker, "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on SW Eng., Vol 14, No.10, October 1988, Pages 1483-1498.*

[Herm1976] P M Herman, "A Data Flow Analysis Approach to Program Testing",*The Australian Computer Journal,* Vol 8, No.3, November 1976, Pages 92-96.

[Hall96] Hall, A *Using Formal Methods to Develop an ATC Information System* IEEE Software, Mar 1996, pp. 66-76.

[Hoa96] Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming." Communications of the ACM **12**(10): pp. 576-580,583.

[Hoff1998] Hoffman, D.M. and Strooper, P.A. *ClassBench: A Methodology and Framework for Automated Class Testing* In *Testing Object-Oriented Software*, Kung, D.C. and Hsia, P. and Gao, J. (ed), pg 152-176, IEEE Computer Society, 1998.

[Jone1990] Jones, C.B. *Systematic Software using VDM* 2nd ed., Prentice Hall, 1990

[King1991] King, K N, Offut, A Jefferson, "A Fortran Language System for Mutation-based Software Testing" *Software - Practice and Experience, Vol. 21 (7), July 1991, Pages 685-718.*

[Knig1985] Knight, J. C. and Amman, P. E. *An experimental evaluation of simple methods for seeding program errors* Proc. 8th International Conference on Software Engineering, London, 1985, pp. 337-342

[Ling94] Linger, R.C. (1994) " *Cleanroom Process Model*" IEEE Software (May 1994) Vol. 11 No. 2 pp.50-58

[Murn01] Murnane, T and Reed, K *On the Effectiveness of Mutation Analysis as a Black Box Testing Techniqu*e Proceedings of the Australian Software Engineering, Conference Canberra, ieee press, August 2001

[Musa1990] Musa, J.D., Iannino, A. and Okumoto (1990) *Software Reliability, Professional Edition,* McGraw-Hill Software Engineering Series.

[Musa1993] J D Musa, "Operational Profiles in Software-Reliability Engineering", *IEEE Software, March 1993, Pages 14-32.*

[Pfle1997] Pfleeger, S.L and Hatton, L *Investigating the Influence of Formal Methods* Ieee-Cs Software Vol 13 February 1997 pp 33-43

[Rich1985] Richardsom, D. J. and Clarke, L. A. *Partition Analysis: A Method Combining Testing and Verification* IEEE Transactions on Software Engineering, Vol. SE-11, no. 12, December 1985, pages 1477 – 1985

[Shoo1983] Shooman, M. *Software Engineering: Design, Reliability and Management,* McGraw-Hill, 1983.

[Sosi97] Sosic, R. and Abramson, D. A. *Guard: A Relative Debugger,* Software Practice and Experience, Vol 27(2), pp 185 – 206 (Feb 1997)

[Stock1996] Stocks, P. and Carrington, D. *A Framework for Specification-Based Testing* IEEE Transactions on Software Engineering, Vol. 22 No. 11, November 1996 pp777-793

[Weis1985] Weiser, M D, Gannon, J D, McMullin, P R,"Comparison of Structural Test Coverage Metrics", *IEEE Software, March 1985, Pages 80 - 85.*

[Whit2001] Whitaker, J *what is software testing? and why is it so hard?* ieee software january/february 2001 pp 70-79

[Zhu1997] Zhu, H., Hall, P. A. V. and May, J. H. R. *Software unit test coverage and adequacy* ACM Computing Surveys December 1997 Volume 29 Issue 4