On the Effectiveness of Mutation Analysis as a Black Box Testing Technique

Tafline Murnane
TATE Associates
Carlton Victoria
Australia
tmurnane@tate.com.au

Associate Professor Karl Reed
Department of Computer Science
and Computer Engineering
La Trobe University Australia
kreed@cs.latrobe.edu.au

Abstract

The technique of mutation testing, in which the effectiveness of tests is determined by creating variants of a program in which statements are mutated, is well known. Whilst of considerable theoretical interest the technique requires costly tools and is computationally expensive. Very large numbers of 'mutants' can be generated for even simple programs.

More recently it has been proposed that the concept be applied to specification based (black box) testing. The proposal is to generate test cases by systematically replacing data-items relevant to a particular part of a specification with a data-item relevant to another. If the specification is considered as generating a language that describes the set of valid inputs then the mutation process is intended to generate syntactically valid and invalid statements. Irrespective of their 'correctness' in terms of the specification, these can then be used to test a program in the usual (black box) manner.

For this approach to have practical value it must produce test cases that would not be generated by other popular black box test generation approaches. This paper reports a case study involving the application of mutation based black box testing to two programs of different types. Test cases were also generated using equivalence class testing and boundary value testing approaches. The test cases from each method were examined to judge the overlap and to assess the value of the additional cases generated. It was found that less than 20% of the mutation test cases for a data-vetting program were generated by the other two methods, as against 75% for a statistical analysis program. This paper analyses these results and suggests classes of specifications for which mutation based test-case generation may be effective.

1 Introduction

Testing software after it is completed remains an important aspect of software quality assurance despite the recent emphasis on the use of formal methods and 'defect-free' software development processes. As has been widely stated, testing does not prove the absence of errors. However, for some classes of programs it is possible in principle to define a 'safe' operational envelope based upon the set of test cases that it processes successfully [1]. Further, clients will frequently write contracts with acceptance testing clauses with the objective of verifying that the software does indeed perform as specified with the intention of taking legal action if it does not. Pre-delivery testing by developers can also provide critical data on the overall effectiveness of the development cycle by identifying residual fault rates.

Over time, a number of specification based (black box or prescriptive) test generation procedures have become popular and have been the subject of numerous studies as to their effectiveness. Broadly speaking, these provide a set of rules of varying detail and clarity that can be applied to a specification to generate test cases.

Traditional mutation analysis is a testing technique that was not originally intended for use with specification based testing. In traditional mutation analysis, a single fault is introduced into the program source code to create a new program version called a 'mutant.' Tests are created and are processed by the original and mutant programs with the goal of causing each mutant to fail (i.e. to produce output that differs from the non-mutant program). The effectiveness of the program test set is evaluated in terms of the number of mutants detected.

Budd and Gopal [2] found it was possible to apply the concept of mutation analysis to specification based testing. Rather than creating mutants from the program source code they are created by mutating the program specification.

In our proposal for mutation analysis, language elements (terminal elements) of the specification are used as mutation substitution elements. Each terminal element is systematically substituted for every other terminal element. A single element substitution produces one mutant specification. A mutation test set is then developed from the mutated specifications.

The goals of this research are:

1. to determine whether or not the mutant tests are able to detect errors in programs and if so, is there a class of specifications that would benefit from this type of testing and,

0-7695-1254-2/01 $10.00 © 2001 IEEE

2. whether this type of testing generates classes of tests that are not produced by other popular forms of black box testing and,
3. whether this type of testing produces small numbers of program-critical tests.

In the case study reported [3], the effectiveness of specification based mutation analysis was compared to boundary value analysis and equivalence class testing. In what follows, we summarise the case study and its results and make suggestions of the classes of programs for which this approach to testing would be effective.

2 Traditional Testing Techniques

2.1 Black Box Testing

The term 'black box' testing is used to describe tests that are derived primarily from a program's specification. In principle, the internal program source code is not considered. Test data derived from the specification is used to systematically test the input and output behaviour of the program. [4]. The goal is to generate a test set that fully exercises the program's functional requirements. Types of testing in this category include equivalence class testing, boundary value analysis, cause-effect graphing, error guessing, model checking and random testing.

2.2 Equivalence Class Testing

Equivalence class testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behavior. Test cases are designed to test the input or output domain partitions. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage [4]. The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed [1].

Jorgensen [5] identified one problem with equivalence partitioning. Often a specification does not define the output for an invalid equivalence class. Tucker [6] also noted that problems occur when the test data chosen for an equivalence class does not represent that partition in terms of the behaviour of the program function that is being tested.

Hamlet and Taylor [7] state that "Partition testing can be no better than the information that defines its sub-domains." If one input in an invalid equivalence class causes a failure in the program then all other inputs in that class must also cause a failure. If this is not the case then the equivalence class is not a good representative of that part of the program and thus the identification of additional partitions may be required. Due to the nature of this approach such problems may not be identified.

2.3 Boundary Value Analysis

Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the 'boundaries' of equivalence classes. Choices of boundary values include above, below and on the boundary of the class.

One disadvantage with boundary value analysis is that it is not as systematic as other prescriptive testing techniques. This is due to the fact that it requires the tester to identify the most extreme values inputs can take. Jorgensen noted that it is this type of abstract thinking that may allow a tester to improve the quality of the test sets used [5].

2.4 White Box Testing

White box testing involves the examination and testing of the program's internal composition. Test data is derived from examining the internal logic, branches and paths of the source code [4]. The goal is either to reach some coverage goal by testing and executing as many paths, branches and statements or other source characteristics as possible [8], or to ensure that certain expressions, decisions, branches, paths or source-attributes are exercised in particular a manner [9]. The number of source-attributes and coverage measures is language dependent and quite large (see for example Wu et al [10]).

White box and black box testing are complimentary and when used together can help to check whether a program conforms to its specification [6] (see for example Offutt and Liu [11][1]).

Rapps and Weyuker [12] noted that as the input domain of a program is generally very large, exhaustive testing is often impractical. Even for a small program containing a limited number of loops and branches, executing every statement is usually infeasible. They furthermore stated that ensuring all paths have been traversed does not guarantee that all errors in the code will have been detected, pointing for example to the problems in detecting 'def-use' errors. This view is supported by Weiser et al [9].

2.5 Traditional (Code-Based) Mutation Analysis

The main objective of traditional (code based) mutation analysis is to determine the effectiveness of a particular test suite. Faults are systematically introduced into the program's source code creating 'flawed clones' of the program called mutants. Each mutant has one language element in a single statement of the original program changed. The element substitution is based on a set of operators called 'mutation operators' [10].

A test case is designed for each mutant to try to detect the 'seeded' error. If the output from the mutated and non-mutated program under this test differs, then the test

[1] Offutt and Liu did state that functional testing had several advantages over structural testing.

has been successful in locating the mutant code and is assumed to be capable of locating similar errors. The mutant is 'killed' and is not executed again against other test cases [13]. Conversely, if the behaviors of the two programs are the same then the error was not detected and the test is discarded. New tests are then designed to try to detect the mutant code. In a complete mutation test all possible mutants of a particular program are produced and tested.

The mutation process may generate changes that leave the mutant functionally equivalent to the original program. This type of mutant should not be killed by any given test case which 'passes' testing the original program. The locating of these 'equivalent mutants' is usually done by hand.

The mutation score is the ratio of the number of killed mutants to the number of non-equivalent mutants and is the measure of the adequacy of the test set. Offutt and Lee stated that a test set is 'mutation-adequate' if the mutation score is 100% [14]. Generally, mutation scores of 90% are difficult to reach and scores over 95% are extremely difficult to achieve [11]. The ultimate goal of mutation analysis is to locate test cases which kill all non-equivalent mutants. Test sets which achieve this are referred to as "adequate relative to mutation" [13].

3 Specification Based Mutation Analysis

Specification based mutation analysis was first suggested by Budd and Gopal in 1984 [2]. Their approach involved mutating formal specifications whose language was defined using predicate calculus. Input test cases were generated by changing operators and predicates of the specification. More recent studies include the use of model checkers to automatically generate specification mutation test sets using several different types of mutation operators (see for example Black et al [16] [17] [18]).

In our case, we treat the specification as a language in which terminal sets can be mutated [3]. A specification can be characterised as a set of language elements which together describe the input and output behavior of a program, in much the same way as the syntax and semantics of the programming language determine valid forms of a program. Each data-item in the specification can be considered as a language or 'terminal' element. Collections of terminal elements are referred to as terminal sets. Production rules define how the terminal elements can be combined.

Substituting one terminal element for another creates one mutant specification. This process is repeated until every terminal element has been substituted for every other terminal element. Since each mutant contains one substituted element it can be referred to as a 'single-defect' mutant. 'Double-defect' mutants can be devised by substituting two terminal elements at a time. 'Production rule mutants' could also be created by mutating the production rules used to generate the input cases.

The 'mutation operator' substitutes one terminal element for another. A simple example is as follows. The terminal set $<terminal_1><terminal_2><termainal_3>$ could create the mutant $<terminal_2><terminal_2><termainal_3>$ by substituting the second terminal element for the first.

One test case is created from each mutant. Mutant test cases are classified as either a 'syntactically valid' or 'syntactically invalid' input. A syntactically valid input would make a program behave in a way that would be expected from a non-mutant input. In an input of this type, the terminal element that was substituted is 'syntactically equivalent' to the terminal element it replaced.

The syntactically invalid class of inputs can be decomposed into 'correct' and 'incorrect'. A syntactically invalid correct input is one that the program should and does recognise as containing a syntactic error. A syntactically invalid incorrect input is one that the program should recognise as containing a syntactic error but does not. This type of input may have located an inadequacy or fault in the program.

Creating a set of double-defect mutants could result in a more rigorous test set, as could production rule mutation. However the number test cases generated could be extremely large. Further, the consequences of the first mutation may directly interfere and complicate the implications of the second mutation, clouding the result of the test.

For some specifications, mutation analysis may produce a test set that appears to resemble a test set produced by random testing. The difference is that mutation analysis produces systematic test sets and is not dependent on randomisation by the tester.

One characteristic that is a requirement of this type of mutation analysis is that the specifications are written in a manner that facilitates the mutation process. It is apparent that some formal or semi-formal method is required where each terminal element is clearly defined. In the case study reported, the use of a semi-formal notation satisfied that requirement.

A shortcoming of mutation analysis is the cost involved in generating and executing test cases and examining the results. It is proposed that this testing technique would benefit greatly from automatic test case generation.

4 Previous Studies on Specification Based Mutation Analysis

Budd and Gopal's [2] approach to specification based mutation analysis involves producing specifications in predicate calculus based upon the predicate structure of the program under consideration. Their notation is chosen so that the input-output relationships are clear. In principle, the specification is mutated so that the new version contains an expression which if true, constitutes an illegal input. The expression should differ from its correct counterpart in that only one element is altered. Special steps are taken to deal with quantifiers, and relational operators may be mutated. An input test case is then produced which meets the mutated specification (i.e. makes it true).

Fabbri, Maldonado, Sugeta and Masiero [15] examined the use of mutation analysis to validate specifications presented as state charts, defining an appropriate mutation operator set to be taken as a fault model. A tool, Proteum/ST, was implemented to support the validation of finite state machine models. The goals of their research were to investigate ways of selecting useful test sets and how to ensure that a specification and its program had been thoroughly tested.

Black, Ammann and Majurski [16] experimented with using a (low-level language) model checker called 'Symbolic Model Version' or SMV to automatically generate complete specification based mutation test sets. "Complete" test sets include inputs and expected results. They used two types of mutation operators, creating both valid and invalid test sets. The model checker was used to produce counterexamples for each mutation operator, where each counterexample was a mutant of the original specification. They noted that their mutation operators were only useful for specifications that were described as finite models (within the context of a model checker). Branch coverage analysis was used to examine the usefulness of the test cases generated, finding that the tests were "quite good, but not perfect." The reported advantages of using a model checker for specification based mutation analysis was that the test case generation was completely automatic, as was the detection of equivalent mutants.

Ammann and Black [17] found that in order to make mutation analysis with a model checker possible they had to decompose specifications to lower language levels. They investigated a way of reducing larger state machines to sub-machines enabling these to be processed by model checkers. This reduction process was referred to as "finite focus." Since model checkers can handle finite state machines of no more than a few thousand states, the specification must allow decomposition. Thus the reduction of the specification's state machine allowed very large software systems to have test cases generated automatically. They proved that finite focus was a sound reduction technique, producing smaller state machines that were valid and creating a smaller mutation adequate test set.

Black, Okun and Yesha [18] examined a method involving the use of the SMV model checker to automatically generate complete mutation test sets from formal specifications using a predefined set of mutation operators. In order to perform the mutation testing, the specification had to be in a form that was readable by SMV. They focussed on redefining and comparing different types of operators and then reducing the number of mutation operators required for good test coverage. They presented classes of operators that provided different levels of coverage (up to 100%) and numbers of mutants created.

Black et al and Budd et al describe complex specification mutation schemes involving conditional logic which will inevitably be reflected in the processing programs. We consider that in many cases the practical advantages can be realised by merely permuting the input specification. Therefore our method of mutation analysis differs from these techniques in the following ways.

1. Only one mutation operator is required making the process far more simple and practical.
2. If the terminal elements are defined then the specification does not have to be changed to fit some predefined format.
3. The more complex the input specification the better the result of testing, for no increase in the complexity of the method.

5 The Case Studies and their Interpretation

The case study involved the comparison of boundary value analysis and equivalence partitioning to specification based mutation analysis [3]. The objective of this comparison was to examine the size and nature of the 'overlap' between the mutation analysis test set and the boundary value and equivalence class test sets. Two semi-formal specifications were used in this approach. Their syntax used a combination of COBOL or PL/I syntax and Backus-Naur Form notation. Both were programming assignments from Software Engineering subjects of La Trobe University (see [19] and [20]).

5.1 The Address Parser Specification

The first specification defines the input for an address parser (data-vetting) program. The input to this program is an address comprised of specific elements, shown in Figure 1. The aim of the program is to parse an address and if it is of a 'correct' format, write it to a file. If not the program is to report which elements of the address are incorrect. The symbols used in the specification are explained in Table 1, while the results of testing are outlined in Table 2. The complete specification included the requirement of directional indicators, for example the address 150 Main Road North Eltham 3095. In the interests of limiting the test set, this variant was not covered.

A standard address:
[{ UNIT }] ∧ddd∧ { , / } ∧ddd∧ <street>∧... <suburb>∧... <postcode>.
FLAT

A special flat/unit address:
[{ UNIT }] ∧ddd∧ <street>∧... <suburb>∧... <postcode>.
FLAT
RSD

A country or care-of address:
[{ C/- }] ∧... <street>∧... <suburb>∧... <postcode>.
C/o

Figure 1 Input elements of the address parser specification.

Table 1 Definition of specification notation. All other symbols are characters included as input.

Symbol	Actual Meaning
∧...	Represents one or more spaces.
d	Represents a digit.
<name>	Represents a character string.
{ }	Select one of the options contained within the braces.
[<name>]	<name> is optional.

Table 2 The results of testing the address parser program.

Number of Tests Created									%of Overlap with Mutation Analysis (MA)	
Mutation Analysis			Equivalence Class Testing (ECT)			Boundary Value Analysis (BVA)				
Total	Passed	Failed	Total	Passed	Failed	Total	Passed	Failed	ECT	BVA
290	10	280	29	10	19	89	31	58	14.5	17.9

A requirement of the address parser program was that if an invalid address was entered then the program has to be capable of recognising the 'incorrect' element(s) and output an appropriate message. This ability is illustrated by test cases one to three of Table 3, which shows sample data and the test methods capable of generating the test cases. The standard address is used as an example in this sample.

Conversely, the output generated by mutation test cases four and five highlight a program fault which was not found by boundary value or equivalence class testing. The fault is that the program produced an output message that did not correctly state which element of the address was incorrect. This illustrates that due to the extreme nature of some of the mutation tests generated, program faults were detected which were not found by conventional black box testing approaches.

Table 3 Sample test data and results of testing the address parser program.

#	Test Case	Program Output	Could be Generated by	Method of Generation
1	UNIT **3095** Main Road Eltham 3095.	Number has too many digits.	MA	Substitute postcode for unit number.
			ECT	Invalid class of unit number.
			BVA	Upper boundary of unit number.
2	UNIT **99** Main Road Eltham 3095.	Number has too few digits.	BVA	Lower boundary of unit number.
			ECT	Invalid class of unit number.
3	UNIT **Test** Main Road Eltham 3095.	Number has too few digits.	ECT	Invalid class of unit number.
4	UNIT **C/o** Main Road Eltham 3095.	Number has too few digits. Space required after suburb. Street not found. Invalid suburb. Full stop not found.	MA	"Care-of" address identifier substituted into the unit number.
5	UNIT 100 **C/o** Eltham 3095.	Space required after suburb. Street not found. Full stop not found. Invalid suburb.	MA	"Care-of" address identifier substituted into the street name.

5.2 The Statistical Analysis Specification

The second specification defines the input of a statistical analysis program which computes the standard deviation and average of values that are tagged by a one-letter identifier. The elements of this specification are listed in Figure 2. The overall results of testing this specification are shown in Table 4. Sample test data and results are shown in Table 5.

Batches of these letters and values are bracketed with the following records.
sbatch∧...<batchno><eor>
and
ebatch∧...<batchno><eor>

The last record in any collection of batches is:
lbatch∧...<eor>|lbatch...<eof>

The records in each batch are of the following form:
<record>::=<lpart><rpart><eor>
<lpart>::=<null>|∧...
<rpart>::=<letter>∧<value>|<rpart>∧<letter>∧<value>
<letter>::= any letter chosen from the set [B-L, S-W, Z]
<value>::= any valid, non-floating point decimal value in the range [-99, 99]

Figure 2 Input elements of the statistical analysis specification.

Table 4 The results of testing the statistical analysis program.

Number of Tests Created									% of Overlap with Mutation Analysis	
Mutation Analysis			Equivalence Class Testing (ECT)			Boundary Value Analysis (BVA)				
Total	Passed	Failed	Total	Passed	Failed	Total	Passed	Failed	ECT	BVA
104	68	36	25	10	15	30	12	18	76	76

Table 5 Sample test data and results of testing the statistical analysis program.

#	Test Case	Program Behaviour	Could be Generated by	Method of Generation
1	sbatch **20** G –**99** ebatch 20	Program accepts the input as valid.	MA	Substitute sbatch number for ebatch number.
			ECT	Valid class of rpart number.
			BVA	Upper boundary of rpart number.
3	sbatch **sbatch** G –99 ebatch 20	Program outputs error message stating that there was no sbatch number found.	MT	sbatch tag substituted into the sbatch number.
4	sbatch **ebatch** G –99 ebatch 20	Program outputs error message stating that there was no sbatch number and the rpart and ebatch tag was not found.	MT	ebatch tag substituted into sbatch number.

5.3 An Examination of the Results

In the specification for the address parser, few terminal elements were syntactically equivalent. Consequently, from the mutation test set produced the program found only a small number of addresses that were syntactically valid. For example the house number could be substituted for the unit/flat number without an error being raised, as both were three digits long. However if the house or unit/flat numbers were swapped with a text sentence such as the street name the program found a syntactic error in the input. The element that was the most interchangeable was the 'space.' One-space markers could be swapped for any one-or-more space markers without errors being detected in the input. The reverse was not equivalent, however the space marker could also be replaced for elements such as all of the optional address elements.

It was found that there was 17.93% equivalence between the mutation analysis and the boundary value analysis test sets, and 14.48% between the mutation analysis and the equivalence class test sets.

For the statistical program there was an extensive overlap between the mutation analysis test set and the boundary value and equivalence class test sets. For example, all three testing methods located errors in inputs involving a missing sbatch or ebatch tag and in inputs containing a letter or value outside of the specified range. Another type of test that produced equivalencies was the replacement of an element with the <null> element. When replacing with the <null> element, the three test sets produced equivalent results in most situations. Therefore there was a large overlap in the tests from the three methodologies.

A 75.96% equivalence was found between the mutation analysis test set and the boundary value and equivalence class test sets.

The testing process showed that although the programs were returning error messages when invalid inputs were entered, in many cases they were not correctly stating which section of the input contained the error. For the statistical program this inadequacy was located by all three testing methodologies. However, for the address parser program most mutation test cases were able to detect these types of errors, whereas the majority of the boundary value and equivalence class tests did not.

6 Mutation Testing Amenable Specifications

In the results reported in the previous section, the address parser specification produced a mutation test set in which there was a modest overlap with the boundary value analysis and equivalence class test sets (less than 20%), while the statistical program's specification produced a substantial overlap in the test sets (75%).

A closer examination of the two specifications suggests that some specifications will be more amenable to mutation based testing than others. While this issue is the subject of future work, we can make some informal comments that will be of practical guidance to practitioners. Consider a simple specification of the following form.

$$\langle terminal_1\rangle\langle sep_1\rangle\langle terminal_2\rangle\langle sep_2\rangle\langle terminal_3\rangle$$

where each of the $\langle sep_i\rangle = \{s_{i_1},\ldots,s_{i_u}\}$
and each of the $\langle terminal_j\rangle = \{t_{j_1},\ldots,t_{j_m}\}$

In general, the nature of the $s_{i_j} \in \langle sep_i\rangle$ and the $t_{j_k} \in \langle terminal_j\rangle$ will be such that it would be unlikely that substituting some arbitrary $t_{3_k} \in \langle terminal_s\rangle$ for $\langle sep_3\rangle$ would produce a test case that would have been generated by either equivalence class testing or boundary value analysis. However, we also need to consider the case of

substituting $t_{3_k} \varepsilon$ **<terminal$_s$>** for **<terminal$_1$>**, which would be a valid mutation operation.

Constructing an equivalence class test requires that there be some basis for dividing the terminal sets (or combinations of them) to construct equivalence partitions. We then choose one element from the partition as a test case. If for some reason the intersection of the terminal sets is non-null then we may have constructed a mutation test by default. However if the terminal sets are distinct then by definition, a valid equivalence class test cannot choose an element from another terminal set. Whether or not invalid equivalence class tests will generate cross-terminal set substitutions depends upon how the terminal set is extended to include illegal values.

In the case of boundary value tests, we point out that if the sets are discrete and finite then the concept of boundaries may have no practical meaning. If they are in some sense continuous or are in a sequence, then boundary values may be considered to exist. Alternatively the boundary values may be stated explicitly. A typical specification for such a terminal might be (without loss of generality) as follows.

<terminal$_i$> ::= $\{R \varepsilon N : ub_i \leq R \leq lb_i\}$

where ub and lb are upper and lower boundaries respectively.

In this case, if there are multiple terminal sets with this definition and their intersection is non-null, then both mutation analysis and boundary value and equivalence class testing can generate test cases that will be identical.

7 Conclusions and Future Work

While specification based mutation analysis can provide a tester with valuable information about the correctness of program behaviour, it is clear that it would not benefit all types of specifications. Future work will include an examination of the feasibility of identifying specifications that will benefit from mutation analysis and the development of mutation operators. Empirical experimentation will determine whether there is a statistical overlap between specification based mutation analysis and other popular forms of black box testing. An additional goal is to investigate whether specification based mutation analysis is effective at producing program-critical tests.

It is also clear that given an appropriate set of (formal) production rules that specify a program's input, a test case generator can be constructed using standard compiler writing techniques. It would then be possible, given appropriate mutation operators, to generate mutant test cases automatically. The simple substitution operator used in the test cases would be straightforward.

Finally, the authors recognise that the approach taken here has properties similar to random test case generation and might generally be regarded as a particular case of this approach.

Acknowledgements

We would like to acknowledge the contributions to this paper of Mr. John Murnane of the Department of Science and Mathematics Education, University of Melbourne, Australia.

We also acknowledge the support of the Department of Computer Science and Computer Engineering at La Trobe University, including the work of Mark Santos, whose use of this technique in a programming assignment lead to our formalisations and to the case study reported here.

Finally, we would like to acknowledge the support of TATE Associates.

References

[1] Karl Reed. *Software Reliability, Testing and Security Class Lecture Notes*. CSE31STM, subject of the Department of Computer Science and Computer Engineering, La Trobe University, Australia, 1998.

[2] Timothy A. Budd, Ajei S. Gopal. Program Testing by Specification Mutation. *Computer Language, vol. 10, no. 1*, Great Britain, 1985, pp. 63-73.

[3] Tafline Murnane. *The Application of Mutation Techniques to Specification Testing*. Honours Thesis, Department of Computer Science and Computer Engineering, La Trobe University, Australia, 1999.

[4] Glenford Myers. *The Art of Software Testing*. Wiley-Interscience Publication, 1979.

[5] Paul Jorgesen. Software Testing: A Craftsman's Approach. *Department of Computer Science and Information Systems, Grand State University Allendale, Michigan and Software Paradigms,* Rockford, Michigan, CRC Press 1995.

[6] Allen Tucker, Robert Cupper, W. Bradley, Richard Epstein, Charles Kelemen. *Fundamentals of Computing II. Abstractions, Data Structures, and Large Software Systems*. McGraw-Hill Inc, 1995.

[7] D. Hamlet, R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, vol. 16, no. 12, December 1990, pp. 1402 – 1411.

[8] Michael Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley & Sons Inc, Canada, 1992.

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

[10] Basili Wu and Karl Reed. A Structure Coverage Tool for ADA Software Systems. *Proceedings of the Joint Ada Conference*, Washington, D.C. (WADAS) March 1987.

[11] A. Offutt, S. Liu. *Generating Test Data from SOFL Specifications*. Preliminary draft yet to be published, written April 1997.

[12] Sandra Rapps, Elaine J Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, vol. SE-11 no. 4 April 1985.

[13] A. Offutt, J. Voas. Subsumption of Conditional Coverage Techniques by Mutation Testing. *Technical Report ISSE-TR-96-01,* January 1996.

[14] A. Offutt, Stephan Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, vol. 20, no. 5, May 1994.

[15] S.C.P.F. Fabbri, J.C. Maldonado, J.C. Sugeta and P.C. Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. *10th International Symposium on Software Reliability Engineering, Proceedings Los Alamitos: IEEE Computer Society*, Boca Raton, USA, pp. 210-219.

[16] Paul Ammann, Paul Black, William Majurski. Using Model Checking to Generate Tests from Specifications. *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods* Brisbane Australia, December 1998, pp 46-54.

[17] Paul Ammann, Paul Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. *Proceedings of the 18th Digital Avionics System Conference*, St. Louis Missouri, October 1999. IEEE vol. 2, section 10.4.6, pp 1-10.

[18] Paul Black, Vadim Okun, Yaacov Yesha. Mutation Operators for Specifications. *15th Annual Software Engineering Conference*, IEEE Computer Society, Grenoble, France, September 2000, pp. 81-88.

[19] Karl Reed. *CSE31STM Assignment Two.* CSE31STM, subject of Department of Computer Science and Computer Engineering, Latrobe University, Australia 1998.

[20] Karl Reed. *CSE32SRT Assignment Two.* CSE32SRT, subject of Department of Computer Science and Computer Engineering, Latrobe University, Australia 1998.