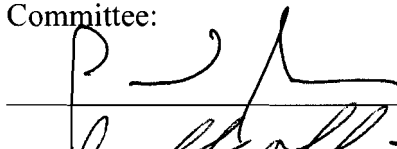


APPLICATIONS OF LOGIC COVERAGE CRITERIA AND LOGIC MUTATION TO
SOFTWARE TESTING

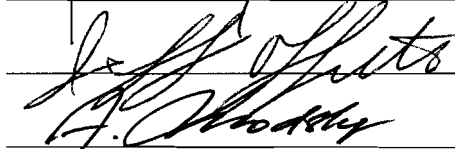
by

Garrett K. Kaminski
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

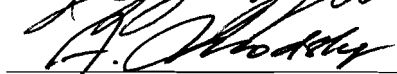
Committee:



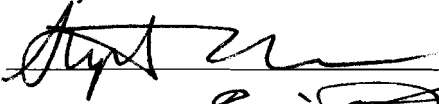
Dr. Paul Ammann, Dissertation Director



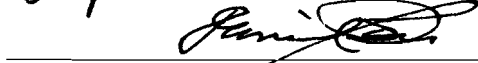
Dr. Jeff Offutt, Committee Member



Dr. Alexander Brodsky, Committee Member



Dr. Stephen Nash, Committee Member



Dr. Daniel Menascé, Senior Associate Dean



Dr. Lloyd J. Griffiths, Dean, The Volgenau
School of Information Technology and
Engineering

Date: 10/18/2010

Fall Semester 2010
George Mason University
Fairfax, VA

Applications of Logic Coverage Criteria and Logic Mutation to Software Testing

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Garrett K. Kaminski
Master of Science
George Mason University, 1998

Director: Dr. Paul Ammann, Associate Professor
The Volgenau School of Information Technology and Engineering

Fall Semester 2010
George Mason University
Fairfax, VA

Copyright: 2010 Garrett K. Kaminski
All Rights Reserved

DEDICATION

This is dedicated to all of the people who have supported both my work and my life over the several years it took me to complete this dissertation. Most especially this includes my wife April who has been a constant source of encouragement, assistance and guidance. This research would not have been completed without her.

ACKNOWLEDGEMENTS

I would like to thank the many people who have supported and motivated my research. Foremost on this list is Dr. Paul Ammann who encouraged me to complete and submit for publication the foundational ideas that led to the majority of this work. I would also like to acknowledge the contributions of the other members of my dissertation committee including Dr. Alexander Brodsky, Dr. Stephen Nash and Dr. Jeff Offutt who have all contributed their knowledge, ideas and encouragement. In addition to my committee, I have also had the privilege of collaborating with other faculty members, fellow students and university researchers. Much of the technical motivation for my research comes from my corporate work at CACI. For this I must thank Jon Jarvis, who has been my technical lead for several years.

TABLE OF CONTENTS

	Page
List of Tables	vii
List of Figures	ix
List of Acronyms	x
List of Theorems	xii
ABSTRACT	xiii
1 Introduction.....	1
1.1 Logic Coverage Criteria Introduction	1
1.2 Mutation Testing Introduction.....	2
1.3 Motivation	3
1.4 Problem Statement	5
1.5 Publications	7
2 Related Work and Background Material	9
2.1 Logic Coverage Criteria Related Work.....	9
2.2 Logic Coverage Criteria Background Material	14
2.3 Mutation Testing Related Work	30
2.4 Mutation Testing Background Material	35
3 Thesis Contributions	36
3.1 Contributions Comparing Minimal-MUMCUT with MUMCUT	41
3.2 Contributions Comparing Minimal-MUMCUT with RACC and RICC	44
3.3 Contributions Comparing TRF-TIF Logic Mutation with Typical Logic Mutation	45
3.4 Contributions Comparing TRF-TIF Logic Mutation with muJava.....	48
3.5 Contributions Comparing TRF-TIF Logic Mutation with SQLMutation	51
3.6 General Threats to Validity	53
4 The Minimal-MUMCUT Criterion.....	55
4.1 Overview of the Minimal-MUMCUT Criterion.	56
4.2 Test Set Size.	59
4.3 Single Minimal DNF Fault Detection.	62
4.4 Double Minimal DNF Fault Detection.....	67
4.5 General Logic Fault Detection.	80
5 Comparison of Minimal-MUMCUT with MUMCUT	94
5.1 Test Set Size (Contribution 1a Parts I and II)	94
5.2 Single Minimal DNF Fault Detection (Contribution 1c).	104
5.3 Double Minimal DNF Fault Detection (Contribution 1d Parts I and II).....	105
5.4 General Logic Fault Detection (Contribution 1e Parts I and II).....	107

6	Comparison of Minimal-MUMCUT with RACC and RICC	109
6.1	Test Set Size (Contribution 2a)	109
6.2	Single Minimal DNF Fault Detection (Contribution 2c Parts I and II).....	116
6.3	Double Minimal DNF Fault Detection (Contribution 2d).....	120
7	TRF-TIF Logic Mutation	122
7.1	Overview of TRF-TIF Logic Mutation.	122
7.2	Mutant Set Size.	138
7.3	Equivalent Mutant Set Size.	139
7.4	Single Minimal DNF Fault Detection.	142
7.5	Double Minimal DNF Fault Detection.....	142
7.6	General Fault Detection.....	142
8	Comparison of TRF-TIF Logic Mutation with Typical Logic Mutation.....	144
8.1	Mutant Set Size (Contribution 3a).....	145
8.2	Equivalent Mutant Set Size (Contribution 3b).....	147
8.3	Single Minimal DNF Fault Detection (Contribution 3c Parts I and II).....	148
8.4	Double Minimal DNF Fault Detection (Contribution 3d Parts I and II).....	153
9	Comparison of TRF-TIF Logic Mutation with muJava.....	155
9.1	Mutant Set Size (Contribution 4a Parts I and II).....	158
9.2	Equivalent Mutant Set Size (Contribution 4b)	162
9.3	Single Minimal DNF Fault Detection (Contribution 4c)	164
9.4	Double Minimal DNF Fault Detection (Contribution 4d).....	167
9.5	General Fault Detection (Contribution 4f)	169
10	Comparison of TRF-TIF Logic Mutation with SQLMutation.....	175
10.1	Mutant Set Size (Contribution 5a).....	178
10.2	Equivalent Mutant Set Size (Contribution 5b)	179
10.3	Single Minimal DNF Fault Detection (Contribution 5c)	180
10.4	Double Minimal DNF Fault Detection (Contribution 5d).....	181
10.5	General Fault Detection (Contribution 5f Parts I and II).	182
11	Conclusion	189
	Appendix A Optimization Model for Selecting NFPs	193
	Appendix B Minimal DNF TCAS Predicates.....	194
	Appendix C General Form TCAS Predicates and Fault Examples	196
	Appendix D Minimal DNF, Minimal CNF and MUMCUT Extension Test Sets	197
	Appendix E RACC Test Set Size Analysis.....	206
	Appendix F RACC and RICC Single Minimal DNF Fault Detection Proof.....	211
	Appendix G RACC Single Minimal DNF Fault Detection Analysis	222
	Appendix H RACC Tests and RACC Fault Detection	227
	Appendix I TRF-TIF Logic Mutations	237
	Appendix J Java Programs and TRF-TIF Mutations	240
	Appendix K Compiere Queries.....	250
	References	280

LIST OF TABLES

Table	Page
Table 1. Basic Definitions	14
Table 2. Minimal DNF Faults	24
Table 3. Relation Between Faults and Criteria.....	25
Table 4. Logic Coverage Criteria Summary.....	29
Table 5. Contribution Summary	38
Table 6. Terms for Reduction in Test Set Size or Mutant Set Size.....	39
Table 7. Terms for Reduction in Fault Detection Capability	40
Table 8. Minimal-MUMCUT Logic Criterion Summary.....	59
Table 9. Criterion Feasibility and LRF Detection	61
Table 10. Minimal-MUMCUT Test Set Size	62
Table 11. Criterion Feasibility Combinations	70
Table 12. Double Fault Detection of Minimal-MUMCUT Tests Based on Criterion Feasibility	71
Table 13. LIF-LIFs Undetected by Minimal-MUMCUT Tests	74
Table 14. Equivalency Relationships Between Faulty and Non-Faulty Predicates	76
Table 15. Equivalent LIF-LIFs as a Percentage of Equivalent LIF Pairings	77
Table 16. Minimal-MUMCUT vs. Minimal-MUMCUT + LIF-LIF Test Set Size.....	79
Table 17. Minimal CNF Logic Criteria Summary	84
Table 18. Minimal DNF Fault Detection	86
Table 19. Minimal CNF Fault Detection.....	87
Table 20. Fault Detection in General Form Boolean Predicates by a Minimal- MUMCUT Test Set based on Minimal DNF, Minimal CNF, or Union	92
Table 21. Minimal-MUMCUT vs. MUMCUT Test Set Size	96
Table 22. Minimal-MUMCUT, MUMCUT and MUMCUT Extension Test Set Sizes.	103
Table 23. Minimal-MUMCUT + LIF-LIF Test Set Size vs. MUMCUT Test Set Size.	106
Table 24. Lowest Bound Maximum RACC Test Set Size Algorithm Results.....	111
Table 25. RACC and Minimal-MUMCUT Test Set Size	114
Table 26. Average Test Set Size for RACC and Minimal-MUMCUT Grouped by Number of Unique Literals.....	116
Table 27. RACC Fault Detection	118
Table 28. RACC Fault Detection Grouped by Number of Unique Literals	119
Table 29. TRF-TIF Faults.....	130
Table 30. Faults for $ab + b\sim c + \sim bc$	137
Table 31. Number of TRF-TIF Logic Mutants Generated	139

Table 32. Number of Typical Logic Mutants Generated	145
Table 33. Faults Produced by Typical Mutation Operators that are not in Lau and Yu's Fault Hierarchy	150
Table 34. Number of Faults Detected by Typical Mutation.....	151
Table 35. MED and MER.....	152
Table 36. TRF-TIF MED and MER	152
Table 37. Arrays and Collections Programs.....	157
Table 38. Number of Software Mutants	158
Table 39. TRF-TIF Mutant Set Size vs. muJava Logic Mutant Set Size.....	161
Table 40. Number and Percentage of Strongly Equivalent Software Mutants.....	162
Table 41. muJava Mutants for Predicate $ab + cd$	166
Table 42. LIF, LRF and TOF for Predicate $ab + cd$	166
Table 43. Number of Strongly Non-Equivalent muJava Mutants Strongly Killed by a Test Set that Weakly Kills all TRF-TIF Logic Mutants	169
Table 44. Number of Query Mutants	178
Table 45. Percentage of Non-equivalent SQLMutation Mutants Killed by a Test Set Killing all TRF-TIF Logic Mutants.....	183
Table 46. Mutation Scores for TRF-TIF Logic Mutation versus a Random Approach .	184
Table 47. Mutation Scores and Number of Database Rows for a SQLFpc Test Set.....	186
Table 48. Mutation Scores and Number of Database Rows for a TRF-TIF Logic Mutation Test Set	186
Table 49. Values of a, b and c in NFPs	207
Table 50. RICC Tests (TOF)	217
Table 51. RICC Tests (LOF)	219
Table 52. RICC Tests (LIF).....	220
Table 53. RICC Tests (LRF)	221

LIST OF FIGURES

Figure	Page
Figure 1. Logic Coverage Criteria, Logic Mutation and Fault Detection	6
Figure 2. Lau and Yu's Fault Hierarchy	27
Figure 3. Subsumption Hierarchy	28
Figure 4. Minimal-MUMCUT Test Set Construction	56
Figure 5. Updated Subsumption Hierarchy with Minimal-MUMCUT	58
Figure 6. Fault Hierarchy based on Infeasibility	67
Figure 7. Updated Subsumption Hierarchy with Minimal CNF Logic Criteria	84
Figure 8. Minimal CNF Fault Hierarchy	86
Figure 9. Extended Fault Hierarchy	133

LIST OF ACRONYMS

Acronym	Full Spelling	Page Reference
ACC	Active Clause Coverage	18
ASF	Associative Shift Fault	91
CACC	Correlated Active Clause Coverage	19
CNF	Conjunctive Normal Form	15
COR	Conditional Operator Replacement	146
CUFNTP	Corresponding Unique False Point Near True Point	82
CUTPNFP	Corresponding Unique True Point Near False Point	22
DNF	Disjunctive Normal Form	15
ENF	Expression Negation Fault	24
ESTF	Expression Stuck at Fault	144
ESTF0	Expression Stuck at Fault 0	150
ESTF1	Expression Stuck at Fault 1	150
GACC	General Active Clause Coverage	19
GICC	General Inactive Clause Coverage	21
ICC	Inactive Clause Coverage	20
LIF	Literal Insertion Fault	25
LNF	Literal Negation Fault	25
LOF	Literal Omission Fault	25
LRF	Literal Reference Fault	25
LSTF	Literal Stuck at Fault	144
LSTF0	Literal Stuck at Fault 0	150
LSTF1	Literal Stuck at Fault 1	150
MCDC	Modified Condition Decision Coverage	9
MED	Mutation Efficiency Difference	136
MER	Mutation Efficiency Ratio	136
MNFP	Multiple Near False Point	22
MNTP	Multiple Near True Point	82
MUFP	Multiple Unique False Point	82
MUMCUT	MUtp-Mnfp-CUTpnfp	23
MUTP	Multiple Unique True Point	21
MUTP/NFP	Multiple Unique True Point / Near False Point	23
NFP	Near False Point	15
NTP	Near True Point	15
ORF.	Operator Reference Fault.	25
ORF.0	Operator Reference Fault.0	150
ORF.1	Operator Reference Fault.1	150
ORF+	Operator Reference Fault+	25
ORF+0	Operator Reference Fault+0	150
ORF+1	Operator Reference Fault+1	150
OTP	Overlapping True Point	16

PCUFPNTP	Partial Corresponding Unique False Point Near True Point	83
PCUTPNFP	Partial Corresponding Unique True Point Near False Point	22
PIF	Parentheses Insertion Fault	91
POF	Parentheses Omission Fault	91
RACC	Restricted Active Clause Coverage	18
RFP	Remaining False Point	16
RICC	Restricted Inactive Clause Coverage	20
SAF	Stuck At Fault	91
SMOTP	Supplementary Multiple Overlapping True Point	73
SQLFpc	SQL Full predicate coverage	175
SVR	Scalar Variable Replacement	146
SVRLOF	Scalar Variable Replacement Literal Omission Fault	150
SVRTOF	Scalar Variable Replacement Term Omission Fault	150
TCAS	Traffic Collision Avoidance System	60
TOF	Term Omission Fault	25
TIF	Term Insertion Fault	3
TIF/LOF	Term Insertion Fault / Literal Omission Fault	122
TIF/LRF	Term Insertion Fault / Literal Reference Fault	122
TNF	Term Negation Fault	25
TOF	Term Omission Fault	25
TRF	Term Reference Fault	3
TRF/LIF	Term Reference Fault / Literal Insertion Fault	122
TRF-TIF	Term Reference Fault – Term Insertion Fault	3
UFP	Unique False Point	15
UOD	Unary Operator Deletion	149
UOI	Unary Operator Insertion	149
UTP	Unique True Point	15
UTPC	Unique True Point Coverage	21
XOR	Exclusive OR	15

LIST OF THEOREMS

Theorem	Page
Theorem 1. Minimal-MUMCUT vs. MUMCUT Single Minimal DNF Fault Detection	104
Theorem 2. Minimal-MUMCUT vs. MUMCUT Double Minimal DNF Fault Detection..	105
Theorem 3. Minimal-MUMCUT vs. RACC/RICC Single Minimal DNF Fault Detection	116
Theorem 4. Minimal-MUMCUT vs. RACC/RICC Double Minimal DNF Fault Detection	120
Theorem 5. TRF-TIF vs. Typical Logic Mutation Equivalent Mutant Set Size.....	147
Theorem 6. TRF-TIF vs. Typical Logic Mutation Single Minimal DNF Fault Detection	148
Theorem 7. TRF-TIF vs. Typical Logic Mutation Double Minimal DNF Fault Detection	153
Theorem 8. TRF-TIF vs. muJava Single Minimal DNF Fault Detection.....	164
Theorem 9. TRF-TIF vs. muJava Double Minimal DNF Fault Detection	167
Theorem 10. TRF-TIF vs. SQLMutation Single Minimal DNF Fault Detection.....	180
Theorem 11. TRF-TIF vs. SQLMutation Double Minimal DNF Fault Detection	181

ABSTRACT

APPLICATIONS OF LOGIC COVERAGE CRITERIA AND LOGIC MUTATION TO SOFTWARE TESTING

Garrett K. Kaminski

George Mason University, 2010

Dissertation Director: Dr. Paul Ammann

Logic is an important component of software. Thus, software logic testing has enjoyed significant research over a period of decades, with renewed interest in the last several years. One approach to detecting logic faults is to create and execute tests that satisfy logic coverage criteria. Another approach to detecting faults is to perform mutation analysis and then find tests that distinguish the original program from each mutant. The fundamental contribution of this dissertation is the development of a new logic coverage criterion and a new logic mutation approach to improve testing in the context of logic expressions in normal form, logic expressions in general form and entire programs. In particular, testing approaches based on current logic coverage criteria and current mutation approaches share the same drawback of not guaranteeing detection of certain logic faults (even when all non-equivalent mutants are killed) and/or are costly in terms of the number of tests required. This dissertation further develops the body of knowledge

in logic coverage criteria and logic mutation testing to address these problems. I show that a new logic coverage criterion can guarantee detecting the same logic faults as current criteria with fewer test cases. I also show that a new logic mutation approach can decrease the number of logic mutants generated while increasing logic fault detection capability. By doing so, a strong theoretical and empirical duality is established between the new logic coverage criterion and the new logic mutation approach.

1 Introduction

1.1 Logic Coverage Criteria Introduction

A common way to test software is to execute tests that satisfy a coverage criterion. A coverage criterion imposes requirements on the tests. For example, one coverage criterion is statement coverage, which demands that every statement in the software be executed by the test set. A logic coverage criterion imposes requirements on tests related to logic expressions (predicates) in source code or other artifacts. For example, one logic coverage criterion is predicate coverage, which requires that each logic expression evaluate to TRUE in at least one test and FALSE in at least one test. Logic coverage criteria differ in fault detection capability and test set size. Many logic coverage criteria exist, but they can be broadly classified into two categories: semantic and syntactic. Semantic criteria make no assumption as to predicate format, whereas syntactic criteria do, with the most common format being minimal Disjunctive Normal Form (DNF).

A common method for evaluating logic coverage criteria is to assess detection of faults in the minimal DNF fault hierarchy of Lau and Yu [30]. Kaminski and Ammann [22] established a complementary minimal Conjunctive Normal Form (CNF) fault hierarchy. In the minimal DNF/CNF context, theoretical analysis can prove that certain faults are guaranteed to be detected. When a predicate is not in minimal CNF or minimal DNF, it can be converted to either normal form and then a syntactic logic coverage

criterion can be applied. However, detection of certain faults in the original predicate is no longer guaranteed by tests that satisfy the syntactic logic coverage criterion. Analysis of fault detection when a predicate is not in minimal CNF or minimal DNF needs to be both theoretical and empirical since some faults are guaranteed to be detected while others are not. Logic coverage criteria can be used to develop tests to detect non-logic faults and in this case the analysis moves completely into the empirical domain. This research introduces a new logic coverage criterion (Minimal-MUMCUT) and evaluates it with respect to test set size and fault detection capability against other logic coverage criteria. The contexts are minimal DNF/CNF logic fault detection, general logic fault detection and general fault detection.

1.2 Mutation Testing Introduction

Mutation testing was originally proposed by DeMillo et al. [11] and Hamlet [14] requires testers to create tests to detect a specified set of faults. Mutant programs that vary from the original program by a single syntactic change are generated. If possible, testers find inputs to distinguish the mutants from the original program to achieve a high mutation score. For a mutant to be distinguished (killed), the statement with the mutation must be reached (reachability), the program state for the mutant must differ from the program state of the original program after the mutated statement is executed (infection) and the difference in program state must propagate to the output (propagation). Mutation score is defined as the number of mutants distinguished (killed) divided by the number of non-equivalent mutants generated. Some mutants are equivalent to the original program in that no input can kill them. The key to mutation testing is the mutation operators used

to make syntactic changes to the source code. In logic mutation, logic mutation operators are used to make syntactic changes to predicates in source code. Mutation testing can also be applied to other artifacts besides source code, but this is outside the scope of this dissertation. For this dissertation, the source code under test is assumed to be deterministic.

Tests created to kill all non-equivalent mutants can be examined theoretically in terms of minimal DNF/CNF logic fault detection, theoretically and empirically in terms of general logic fault detection and empirically in terms of general fault detection. Kaminski and Ammann [18] introduced a new logic mutation approach known as TRF-TIF (Term Replacement Fault – Term Insertion Fault) mutation to produce solely selective logic mutants. In order to kill the mutants generated by this tool it is necessary (but not sufficient) to satisfy the Minimal-MUMCUT logic coverage criterion. Thus, TRF-TIF logic mutation subsumes the Minimal-MUMCUT logic coverage criterion. The TRF-TIF tool uses novel mutation operators whose corresponding fault types sit atop an extended minimal DNF logic fault hierarchy. This research evaluates mutant set size, equivalent mutant set size and fault detection capability for the TRF-TIF logic mutation approach in comparison to other mutation approaches. This research also examines the degree to which killing all TRF-TIF logic mutants kills general mutants.

1.3 Motivation

The motivation for this research has two parts, one based on logic coverage criteria and the other based on logic mutation. In summary, testing approaches based on current

logic coverage criteria or current mutation approaches share the drawback of not guaranteeing detection of certain logic faults (even when all mutants are killed) and/or are costly in terms of the number of tests required.

Logic coverage criteria exist that require small test set size, but they do not guarantee detecting common logic faults. Conversely, logic coverage criteria exist that guarantee detecting common logic faults, but these criteria require a large test set size. Part of the reason for this is that current logic coverage criteria do not handle infeasibility efficiently, which in turn results in unnecessary tests in that all faults in Lau and Yu's fault hierarchy can still be detected even when one or more tests are removed.

Current logic mutation approaches have several problems. One, mutants that are syntactically different from each other yet semantically the same can be generated. Two, mutants are generated that are guaranteed to be killed by a test that kills some other generated mutant. While Offutt, Rothermel and Zapf [37] showed that mutation testing can use selective logic mutation operators to offset this cost, selective logic mutation operators in current mutation tools are used inefficiently. Three, current mutation tools lack logic mutation operators that generate mutants that, when killed, guarantee killing the most number of other mutants. These inefficiencies cause excess mutants to be generated and reduce fault detection capability.

These drawbacks call for new logic coverage criteria and new logic mutation approaches and this research addresses exactly these drawbacks.

1.4 Problem Statement

The main logic coverage criterion problem that is addressed is how to best achieve a balance of reducing the number of tests needed to detect logic faults while at the same time increasing the number of logic faults detected, regardless of predicate format. The main logic mutation problem that is addressed is how to reduce the number of logic mutants generated while increasing fault detection (assuming all non-equivalent mutants are killed). A secondary mutation problem that is addressed is how to reduce the number of equivalent mutants. The above problems are addressed by extending a current logic fault hierarchy, inventing new logic mutation operators that sit atop the extended logic fault hierarchy, providing partial solutions to the equivalent mutant problem and analyzing logic coverage criterion feasibility at a low level of detail. Figure 1 displays the problems this research addresses at a high level.

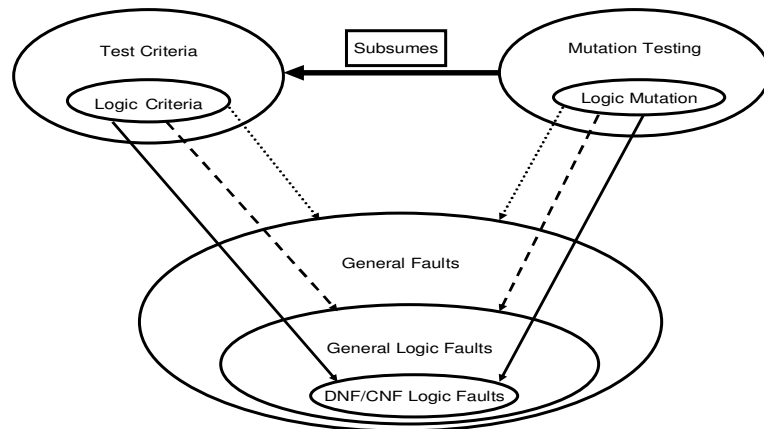


Figure 1 Logic Coverage Criteria, Logic Mutation and Fault Detection

The two thin solid arrows in Figure 1 represent the problem of increasing the number of minimal DNF/CNF logic faults guaranteed to be detected while decreasing test set or mutant set size. These arrows point to logic faults for predicates in minimal DNF/CNF. The fact that the arrows are solid indicates that guaranteed fault detection can be proven. In other words, theoretical analysis is applied.

The two thin dashed arrows in Figure 1 represent the problem of increasing the number of general logic faults detected while decreasing test set or mutant set size, but without a guarantee of detection. These arrows point to logic faults for predicates regardless of format. The fact that the arrows are dashed indicates that some types of faults can be guaranteed to be detected theoretically but that others cannot and thus require empirical study.

The thin dotted arrows in Figure 1 represent the problem of increasing the number of faults (logic and non-logic) detected while decreasing test set or mutant set size, but without a guarantee of detection. These arrows point to faults in general. The fact that these arrows are dotted indicates that non-logic fault detection relations cannot be guaranteed theoretically but rather require empirical study.

The thick arrow between mutation testing and test criteria represents that a subsumption relationship can exist. That is, depending on the mutation operators used, mutants can be generated such that a test set that kills all non-equivalent mutants is guaranteed to satisfy a specific criterion. TRF-TIF logic mutation subsumes the Minimal-MUMCUT logic coverage criterion because in order to kill all TRF-TIF logic mutants, Minimal-MUMCUT must be satisfied.

This dissertation is organized as follows. Chapter 2 describes related work and background material. Chapter 3 summarizes the thesis contributions. Chapter 4 provides an overview of the Minimal-MUMCUT criterion and Chapter 7 provides an overview of TRF-TIF logic mutation. Chapters 5, 6 and 8-10 discuss theoretical and empirical results. Chapter 11 draws conclusions.

1.5 Publications

Material in this dissertation is used with permission and has been published in the following:

Refereed Journals

- [22] G. Kaminski and P. Ammann. Reducing Logic Test Set Size While Preserving Fault Detection. To appear in *Software Testing, Verification, and Reliability*. Wiley.
- [23] G. Kaminski, U. Praphamontripong, P. Ammann and J. Offutt. A Logic Mutation Approach to Selective Mutation Using Programs and Queries. Accepted with minor revisions by *Information and Software Technology, Special Issue on Mutation Testing*.
- [24] G. Kaminski, G. Williams and P. Ammann. Reconciling Perspectives of Logic Testing for Software. *Software Testing, Verification and Reliability*, 18(3):149-188, September 2008. Wiley.

Refereed Conferences and Workshops

- [18] G. Kaminski and P. Ammann. Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing. *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Pages 386-395. Denver, CO. April, 2009.
- [19] G. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection. *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Pages 167-176. Denver, CO. April, 2009.

- [20] G. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Double Fault Detection. *Proceedings of the Mutation Workshop at the 2nd International Conference on Software Testing, Verification and Validation*. Denver, CO. April, 2009.

Other

- [21] G. Kaminski and P. Ammann. Applying MCDC to Large DNF Expressions. *Proceedings of the 9th International Conference on Software Engineering Research and Practice*. Las Vegas, NV. July, 2010.

Additional publications by the author that are related to the dissertation, but are not formally included because they are outside the scope of comparing logic coverage criteria and logic mutation approaches:

Refereed Conferences and Workshops

G. Kaminski and P. Ammann. Applications of Optimization to Logic Testing. *Proceedings of the Constraints in Software Testing, Verification and Analysis Workshop at the 3rd International Conference on Software Testing, Verification and Validation*. Paris, France. April, 2010.

Other

G. Kaminski, U. Praphamontripong, P. Ammann and A.J. Offutt. An Evaluation of the Minimal-MUMCUT Logic Criterion and Prime Path Coverage. *Proceedings of the 9th International Conference on Software Engineering Research and Practice*. Las Vegas, NV. July, 2010.

2 Related Work and Background Material

In this chapter, related work and background is discussed. Section 2.1 summarizes logic coverage criteria related work. Section 2.2 focuses on logic coverage criteria background material. Section 2.3 summarizes mutation testing related work. Section 2.4 focuses on mutation testing background material.

2.1 Logic Coverage Criteria Related Work

A logic coverage criterion imposes requirements on tests related to logic expressions in source code or other artifacts. Logic coverage criteria differ in fault detection capability and test set size. Many such criteria exist, but they can be broadly classified into two categories: semantic and syntactic. Semantic criteria make no assumption as to predicate format, whereas syntactic criteria do, with the most common format being minimal DNF. Details of various logic coverage criteria are presented next, starting with semantic criteria followed by syntactic criteria. This section concludes with an overview of logic fault detection.

Semantic Logic Coverage Criteria

Chilenski and Miller [10] discuss the modified condition decision coverage (MCDC) criterion, which is the best known semantic logic coverage criterion. Chilenski and Miller [10] differentiate between Weak and Strong MCDC. Weak MCDC treats multiple occurrences of the same literal (condition) as one occurrence. Strong MCDC treats

multiple occurrences of the same literal (condition) as different occurrences. That is, in Weak MCDC tests are formed on the basis of each unique literal whereas in Strong MCDC tests are formed on the basis of each literal. Chilenski [9] further differentiates MCDC into Unique-Cause MCDC, Unique-Cause + Masking MCDC and Masking MCDC. Unique-Cause MCDC requires all other conditions to be fixed when varying the condition of interest from TRUE to FALSE. Masking MCDC allows other conditions to vary when varying the condition of interest from TRUE to FALSE. Unique-Cause + Masking MCDC requires all other conditions to be fixed when varying the condition of interest from TRUE to FALSE, unless an infeasibility arises, in which case other conditions can vary so as to remove the infeasibility. Thus, six flavors of MCDC exist: Strong Unique-Cause MCDC, Strong Unique-Cause + Masking MCDC, Strong Masking MCDC, Weak Unique-Cause MCDC, Weak Unique-Cause + Masking MCDC and Weak Masking MCDC. When referring to MCDC, any of the three versions of weak MCDC is implied unless otherwise stated. Ammann and Offutt [2] discuss five related semantic criteria: General Active Clause Coverage (GACC), Correlated Active Clause Coverage (CACC), Restricted Active Clause Coverage (RACC), General Inactive Clause Coverage (GICC) and Restricted Inactive Clause Coverage (RICC). RACC corresponds to Weak Unique-Cause MCDC and CACC corresponds to Weak Masking MCDC. Like MCDC tests, ACC and ICC tests can fail to detect most faults in Lau and Yu's fault hierarchy [24]. This dissertation establishes this fact in section 6.2.

Syntactic Logic Coverage Criteria

Weyuker, Goradia and Singh [46] proposed the MAX-A and MAX-B syntactic criteria, whose tests guarantee detecting all faults in the hierarchy. Chen, Lau and Yu [8] developed the MUTP (Multiple Unique True Point), MNFP (Multiple Near False Point) and CUTPNFP (Corresponding Unique True Point Near False Point) criteria and integrated them into the MUMCUT (MUTP-MNFP-CUTPNFP) criterion, whose tests are guaranteed to detect all faults in the hierarchy with a smaller test set size. Assuming minimal DNF, the CUTPNFP criterion and Strong Unique-Cause MCDC are identical as are the UTPNFP criterion and Strong Masking MCDC [24]. Chen and Lau [6] implemented the MUTP Greedy algorithm to satisfy the MUTP criterion as a constituent of the MUMCUT criterion. Yu and Lau [50] showed how MUMCUT test set size can vary depending on which heuristic is used to generate the test set. Kaminski, Williams and Ammann [24] proposed the MUTP/NFP criterion, whose tests are guaranteed to detect all faults in the hierarchy while further reducing test set size, but only if the criterion is feasible. Sun et al. [41] analyzed how the MUMCUT criterion can be extended to apply to predicates in any format. Kaminski and Ammann [19] proposed the Minimal-MUMCUT criterion, which reduces MUMCUT test set size without sacrificing fault detection regardless of feasibility. This dissertation develops Minimal-MUMCUT in chapter 4 and evaluates its fault detection capability and test set size with respect to MUMCUT in chapter 5.

Logic Fault Detection

The seminal work in composing a logic fault hierarchy was performed by Kuhn [26]. Okun, Black and Yesha [38] showed how Kuhn’s logic fault hierarchy can apply to predicates in any format. Lau and Yu [30] refined Kuhn’s work by introducing new faults and detection relationships assuming minimal DNF. Kaminski and Ammann [18] extended Lau and Yu’s fault hierarchy to include new fault types that correspond to mutation operators in typical logic mutation approaches and new highly selective logic mutation operators not in typical logic mutation approaches. (Typical logic mutation refers to a hypothetical tool including a common set of mutation operators.)

Fault detection guarantees by syntactic tests that hold for minimal DNF predicates do not hold for non-minimal DNF predicates. This raises three issues that prior researchers have investigated. One, how well do tests based on minimal DNF detect faults in non-minimal DNF predicates? Two, what types of software have a high percentage of their predicates in minimal DNF? Three, what extensions are necessary to expand fault detection when the minimal DNF assumption fails to hold? For the first issue, Yu and Lau [48] found that of a sample of 20 non-minimal DNF predicates, over 99% of seeded faults were detected by MUMCUT tests formed from the minimal DNF version of the predicates. This dissertation shows in section 5.4 that over 98% of seeded faults in non-minimal DNF predicates were detected by Minimal-MUMCUT tests formed from the minimal DNF versions of the predicates. For the second issue, Chilenski [9] found that 95% of 20,256 predicates in avionics software were in minimal DNF. This dissertation reports results in section 5.4 showing that 85% of predicates in this sample that contained

at least 3 unique literals were in either minimal DNF or minimal CNF. For the third issue, Sun et al. [41] analyzed what patterns of faults the MUMCUT criterion can miss for general form predicates and how the MUMCUT criterion can be extended (MUMCUT extensions) to guarantee detection of these faults. In this dissertation a comparison of Minimal-MUMCUT vs. MUMCUT extension test set size is given in section 5.1.

Double Logic Fault Detection

A double logic fault occurs when two faults are introduced into a predicate. Two faults can be coupled together such that inputs detecting each in isolation cannot detect the corresponding double fault. Fault coupling rarely occurs [16, 31] as inputs that detect each fault usually detect the double fault. However, double faults are more likely to occur than higher order faults based on the competent programmer hypothesis [11] (which states that competent programmers write programs that differ from a correct version by relatively few simple faults). Offutt [32] investigated fault coupling using an empirical approach. He found that tests that detected all single faults seeded into a program detected 99.9% of double faults. Thus, he concluded fault coupling rarely occurs. How Tai Wah [16] examined fault coupling from a theoretical perspective. He analyzed the ratio of the number of tests that detect single faults but not the corresponding double faults to the number of tests that detect single faults. He showed that the ratio is small and concluded fault coupling rarely occurs. Lau et al. [27, 28, 29] studied logic fault coupling. They showed that MUMCUT tests, which detect all single faults in the hierarchy, guarantee detection of all but 8 of 92 double fault types. They proposed additional criteria to guarantee double fault detection and list conditions necessary to

detect each double fault. However, these conditions are not specified in terms of the criterion feasibility of the MUTP, CUTPNFP, and MNFP criteria which compose MUMCUT. This dissertation compares Minimal-MUMCUT and MUMCUT double fault detection in section 5.3.

2.2 Logic Coverage Criteria Background Material

In this dissertation, Minimal-MUMCUT is compared with other logic criteria. In this section, background material is presented for these other logic criteria. First, some basic definitions are presented. Next, several semantic logic coverage criteria are described, followed by a description of several syntactic logic coverage criteria. Several Minimal DNF logic faults are then examined, followed by background material on subsumption. The section ends with a summary containing a table that summarizes each of the logic criteria presented in this section. Table 1 lists the definitions for various logic terms and symbols used throughout this section and the rest of this dissertation.

Table 1 Basic Definitions

Term or Symbol	Definition
1	The Boolean value TRUE.
0	The Boolean value FALSE.
Literals	Variables representing clauses in a predicate.
+	OR operator.
Adjacency between literals or parentheses	AND operator.
XOR	Exclusive OR operator.
~, !	Negation (also indicated by a – above a literal or term).
Term	A set of literals.

Term or Symbol	Definition
Disjunctive Normal Form (DNF)	Predicate syntax where terms are separated by OR and literals are separated by AND. For example, $ab + cd$.
Conjunctive Normal Form (CNF)	Predicate syntax where terms are separated by AND and literals are separated by OR. For example, $(a + b)(c + d)$.
DNF implicant	A term that when TRUE, means the predicate is TRUE. For example, ab is a DNF implicant in $ab + cd$.
CNF implicant	A term that when FALSE, means the predicate is FALSE. For example, $(a + b)$ is a CNF implicant in $(a + b)(c + d)$.
DNF prime implicant	DNF implicant where removing a literal could potentially change the predicate value. For example, ab is a DNF prime implicant in $ab + cd$ but $ab!c$ is not a DNF prime implicant in $ab!c + abc$.
CNF prime implicant	CNF implicant where removing a literal could potentially change the predicate value. For example, $(a + b)$ is a prime CNF implicant in $(a + b)(c + d)$ but $(a + b + !c)$ is not a CNF prime implicant in $(a + b + !c)(a + b + c)$.
Irredundant DNF	Predicate syntax where it is possible to make each term TRUE in turn while all other terms are FALSE.
Irredundant CNF	Predicate syntax where it is possible to make each term FALSE in turn while all other terms are TRUE.
Minimal DNF	Predicate syntax in irredundant DNF where all implicants are DNF prime implicants.
Minimal CNF	Predicate syntax in irredundant CNF where all implicants are CNF prime implicants.
Unique True Point (UTP)	An assignment of values to literals in a minimal DNF predicate such that only a single term is TRUE. In $ab + cd$, UTPs for ab are 1100, 1101, 1110.
Unique False Point (UFP)	An assignment of values to literals in a minimal CNF predicate such that only a single term is FALSE. In $(a + b)(c + d)$, UFPs for $(a + b)$ are 0001, 0010, 0011.
Near False Point (NFP)	An assignment of values to literals in a minimal DNF predicate such that the predicate is FALSE but negating a single literal makes the predicate TRUE. In $ab + cd$, NFPs for a are 0100, 0101, 0110.
Near True Point (NTP)	An assignment of values to literals in a minimal CNF predicate such that the predicate is TRUE but negating a single literal makes the predicate FALSE. In $(a + b)(c + d)$, NTPs for a are 1001, 1010, 1011.

Term or Symbol	Definition
Corresponding NFP	In a minimal DNF predicate, an NFP that differs from a UTP for the literal's term only in the value of that literal. In $ab + cd$, 0100 is a corresponding NFP for literal a as it differs from the UTP 1100 for term ab only in the value of literal a .
Corresponding NTP	In a minimal CNF predicate, an NTP that differs from a UFP for the literal's term only in the value of that literal. In $(a + b)(c + d)$, 1011 is a corresponding NTP for literal a as it differs from the UFP 0011 for term $(a + b)$ only in the value of literal a .
Overlapping True Point (OTP)	In a minimal DNF predicate, an OTP is an assignment of values to literals such that at least two of the terms are TRUE. In $ab + cd$, 1111 is an OTP.
Remaining False Point (RFP)	In a minimal DNF predicate, a RFP is an assignment of values to literals such that the predicate evaluates to FALSE but the assignment does not represent an NFP. In $ab + cd$, 0000 is a RFP.
Feasible	A logic coverage criterion is feasible if and only if it is possible to construct all required tests.
Combinatorial Coverage	Demands for an exhaustive test set. In other words, the test set must include all possible combinations of the values of literals in a given predicate.

Exhaustive logic test size grows exponentially, requiring tests of $O(2^n)$, where n is the number of unique literals in the predicate. Thus, testers have invented less expensive criteria. Several criteria are described next, starting with semantic flavors and then onto syntactic flavors. For all of these criteria, if an infeasibility occurs the tests chosen should satisfy the requirements as fully as possible.

Semantic Logic Coverage Criteria

The general idea behind the ACC tests is to evaluate under what conditions each unique literal determines the outcome of the predicate. In other words, each variable will become the final decision for whether the predicate will be TRUE or FALSE. A key

feature of the semantic domain is that the criteria are independent of predicate form. Also, each literal is not necessarily unique as literals repeated in different terms are not treated as unique. Definitions of several semantic criteria are given below. In these definitions, a *test pair* represents two distinct test points, each with its own specific assignment of Boolean values to all the literals in a given predicate. Two distinct test points are paired because ACC requires pairs of points for literals. To understand the definitions, it is first necessary to understand how a literal determines a Boolean function.

Literal Determination of a Boolean Function

An important concept is the idea of a literal determining the value of a Boolean function. The Boolean derivative [24] can be used for this reason and is defined as $f_x = f_{x=1} \oplus f_{x=0}$, where $f_{x=1}$ is the value of f when literal $x = 1$ and $f_{x=0}$ is the value of f when literal $x = 0$. If the Boolean function is written in terms of literal x , then

$$f = \bigcup_{i=1}^n a_i + x \bigcup_{i=1}^m b_i + \bar{x} \bigcup_{i=1}^k c_i \text{ and the Boolean derivative is:}$$

$$f_x = f_{x=1} \oplus f_{x=0}$$

$$f_x = [\bigcup_{i=1}^n a_i + 1 \bigcup_{i=1}^m b_i + 0 \bigcup_{i=1}^k c_i] \oplus [\bigcup_{i=1}^n a_i + 0 \bigcup_{i=1}^m b_i + 1 \bigcup_{i=1}^k c_i] \quad f_x = [\bigcup_{i=1}^n a_i + \bigcup_{i=1}^m b_i] \oplus [\bigcup_{i=1}^n a_i + \bigcup_{i=1}^k c_i]$$

$$f_x = \prod_{i=1}^n \bar{a}_i [\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i]$$

Likewise, one can evaluate when the literal x does not determine the outcome of Boolean function f . To compute this case, negate the Boolean Derivative to obtain:

$$\bar{f}_x = \prod_{i=1}^n \bar{a}_i \left[\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i \right]$$

$$\begin{aligned}\overline{f_R} &= \bigcup_{i=1}^n a_i + \left[\prod_{i=1}^m \overline{b_i} + \bigcup_{i=1}^k c_i \right] \left[\bigcup_{i=1}^m b_i + \prod_{i=1}^k \overline{c_i} \right] \\ \overline{f_R} &= \bigcup_{i=1}^n a_i + \bigcup_{i=1}^m b_i \bigcup_{i=1}^k c_i + \prod_{i=1}^m \overline{b_i} \prod_{i=1}^k \overline{c_i}\end{aligned}$$

Active Clause Coverage (ACC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j determines the Boolean value of f and c_j is 0 and 1. There are three distinct flavors of ACC as described below.

Restricted Active Clause Coverage (RACC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j determines the Boolean value of f , c_j is 0 and 1 and all other literals c_i remain constant. By definition, f will be 0 in one test and 1 in the other test, so predicate coverage is satisfied. RACC selects a corresponding UTP-NFP pair for each unique literal (as opposed to each literal) when the predicate is in minimal DNF. Thus, if a literal repeats the repeating instances do not require a corresponding UTP-NFP pair. Consider $f = ab + cd$. Literal a determines f when $b=1, c=1, d=0$ or when $b=1, c=0, d=1$, or when $b=1, c=0, d=0$. Thus, RACC tests for a could be 1110 and 0110 since the values of b, c and d need to remain constant. Likewise, RACC tests for b could be 1110 and 1010. Literal c determines f when $a=0, b=1, d=1$ or when $a=1, b=0, d=1$, or when $a=0, b=0, d=1$. Thus, RACC tests for c could be 0111 and 0101 since the values of a, b and d need to remain constant. Likewise, RACC tests for d could be 0111 and 0110. A test set is {1110, 0110, 1010, 0111, 0101}. Now consider $f = ab + ac$ and note that literal a repeats. Literal a determines f as long as b and c are not both 0. Thus, for literal a RACC tests could be 110 and 010. Neither of the points is an NFP for literal a in term ac . RACC is identical to Weak Unique-Cause

MCDC, which is a standard of the United States Federal Aviation Administration for safety critical software in commercial aircraft.

Correlated Active Clause Coverage (CACC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j determines the Boolean value of f , c_j is 0 and 1 and f is 0 and 1, respectively (so predicate coverage is satisfied). CACC selects a UTP and NFP for each unique literal (as opposed to each literal) when the predicate is in minimal DNF. Thus, if a literal repeats the repeating instances are ignored. The UTP and NFP chosen do not need to be a corresponding pair as in RACC. In other words the UTP and NFP chosen do not have to differ only in the value of the literal of interest. That is, other literals in the predicate may flip values between the UTP and NFP. Consider $f = ab + cd$. CACC tests for a could be 1110 and 0100 since the value of c does not need to remain constant. Likewise, CACC tests for b could be 1110 and 1000. CACC tests for c could be 0111 and 0001 since the value of b does not need to remain constant. Likewise, CACC tests for d could be 0111 and 0010. A test set is {1110, 0100, 1000, 0111, 0001, 0010} although the RACC test set above would also suffice as any RACC test set satisfies CACC. CACC is identical to Weak Masking MCDC.

General Active Clause Coverage (GACC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j determines the Boolean value of f and c_j is 0 and 1. Note that it is not explicitly required that f evaluates to 0 in one test and 1 in the test, so predicate coverage is not necessarily satisfied. Although predicate coverage will usually be satisfied, it is possible to satisfy GACC when f evaluates to only

0 or 1. The requirements for GACC and ACC are the same. Any RACC or CACC test set is a GACC test set.

Inactive Clause Coverage (ICC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1 and f is 0 for both elements of the test pair. There is also a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1 and f is 1 for both elements of the test pair. There are two distinct flavors of ICC as described below, each of which corresponds to a flavor of Reinforced Condition Decision Coverage (RCDC) as described by Vilkomir and Bowen [45].

Restricted Inactive Clause Coverage (RICC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1, all other literals c_i remain constant and f is 0 for both elements of the test pair. There is also a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1, all other literals c_i remain constant and f is 1 for both elements of the test pair. Consider $f = ab + cd$. Literal a does not determine f when $b=0$ or when $c=1$ and $d=1$. Thus, RICC tests for a could be 1000 and 0000 since the values of b , c and d need to remain constant. Likewise, RICC tests for b could be 0100 and 0000. Literal c does not determine f when $d=0$ or when $a=1$ and $b=1$. Thus, RICC tests for c could be 0010 and 0000 since the values of a , b and d need to remain constant. Likewise, RICC tests for d could be 0001 and 0000. A test set is {1000, 0100, 0001, 0001, 0000}.

General Inactive Clause Coverage (GICC) [2]: Given a Boolean function, f , composed of literals c_j , there is a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1 and f is 0 for both elements of the test pair. There is also a test pair for each c_j such that c_j does not determine the Boolean value of f , c_j is 0 and 1 and f is 1 for both elements of the test pair. The requirements for GICC and ICC are the same. Every RICC test set is a GICC test set.

Syntactic Logic Coverage Criteria

Syntactic criteria require a predicate to be in a particular format. For all of the syntactic criteria below, the format is assumed to be minimal DNF. Also, syntactic criteria treat literals that repeat in different terms as distinct.

Unique True Point Coverage (UTPC) [2]: Given minimal DNF Boolean functions, f and \bar{f} , terms p_i in f and terms p_k in \bar{f} , there is a test for a UTP for each p_i in f and p_k in \bar{f} . Consider $f = ab + cd$. 1100 is a UTP for term ab and 0011 is a UTP for term cd . $\bar{f} = \sim a \sim c + \sim a \sim d + \sim b \sim c + \sim b \sim d$. 0101 is a UTP for term $\sim a \sim c$, 0110 is a UTP for term $\sim a \sim d$, 1001 is a UTP for term $\sim b \sim c$ and 1010 is a UTP for term $\sim b \sim d$.

Multiple Unique True Point (MUTP) [8]: Given a minimal DNF predicate, form tests for a UTP for each term such that all literals not in the term attain values 1 and 0. Consider $f = ab + cd$. A UTP for the first term must have $a=1, b=1$. Tests for c and d to each equal 0 and 1 are 1101 and 1110. A UTP for the second term must have $c=1$ and $d=1$. Tests for a and b to each equal 0 and 1 are 0111 and 1011. A test set is {1101, 1110, 0111, 1011}.

Multiple Near False Point (MNFP) [8]: Given a minimal DNF predicate, form tests for an NFP of each literal such that all literals not in the literal's term attain values 1 and 0. Consider $f = ab + cd$. NFPs for a and b so that c and d each equal 0 and 1 are 0101, 0110, 1001, and 1010. NFPs for c and d so that a and b each equal 0 and 1 are 0101, 1001, 0110, and 1010. A test set is {0101, 0110, 1001, 1010}.

Corresponding Unique True Point Near False Point (CUTPNFP) [8]: Given a minimal DNF predicate, for each literal find a UTP and NFP such that only the literal changes value (all other literals must be fixed). Consider $f = ab + cd$. A UTP for the first term must have $a=1, b=1$. If $c=0$ and $d=1$, tests for the literals in ab are 1101, 0101, and 1001. A UTP for the second term must have $c=1, d=1$. If $a=1$ and $b=0$, tests for the literals in cd are 1011, 1001, and 1010. A test set is {1101, 0101, 1001, 1011, 1010}. When a predicate is in minimal DNF, CUTPNFP is equivalent to Strong Unique-Cause MCDC. When a minimal DNF predicate is a Singular Boolean Expression (meaning each literal occurs only once), CUTPNFP is equivalent to RACC (and hence Weak Unique-Cause MCDC).

Partial-Corresponding Unique True Point Near False Point (PCUTPNFP) [22]: Given a minimal DNF predicate, for each literal find a UTP and NFP such that the literal changes value and the only literals that must be fixed are literals that must be fixed in a UTP for the term of interest. This criterion is more flexible than CUTPNFP and is subsumed by it (any CUTPNFP test set is a PCUTPNFP test set). Consider $f = ab + cd$. For term ab , a MUTP test set is {1101, 1110}. To satisfy CUTPNFP for literal a in term ab , a corresponding NFP of 0101 or 0110 must be chosen. However, PCUTPNFP permits

0100 to be chosen as the NFP. 0100 differs from each UTP in either a and c or a and d . Thus, 0100 is not a corresponding NFP but it can still be chosen to satisfy PCUTPNFP because literals c and d can be 0 or 1 in a UTP for term ab . While PCUTPNFP does not offer test set size savings over CUTPNFP for the example of $ab + cd$, it can for other examples because it allows greater flexibility in choosing NFPs so that they can overlap for literals in different terms.

MUTP/MNFP/CUTPNFP (MUMCUT) [8]: Given a minimal DNF predicate, satisfy the MUTP, MNFP and CUTPNFP criteria. Consider $f = ab + cd$. 1101 and 1110 are UTPs for ab . 0101 and 0110 are NFPs for a that differ from a UTP for ab only in the value a . 1001 and 1010 are NFPs for b that differ from a UTP for ab only in the value of b . 0111 and 1011 are UTPs for cd . 0101 and 1001 are NFPs for c that differ from a UTP of cd only in the value of c . 0110 and 1010 are NFPs for d that differ from a UTP for cd only in the value of d . In the NFPs above each literal not in the term of interest attains 1 and 0 so the MNFP criterion is satisfied. A test set is {1101, 1110, 0101, 0110, 1001, 1010, 0111, 1011}.

Multiple Unique True Point / Near False Point (MUTP/NFP) [24]: Given a minimal DNF predicate, satisfy the MUTP criterion for each term and select an NFP for each literal. The NFP chosen for a given literal does not need to differ from a selected UTP only in the value of the given literal. When the MUTP criterion is feasible or when the predicate is a singular Boolean expression, MUTP/NFP tests guarantee the detection of all faults in Lau and Yu's fault hierarchy and do so with a potentially smaller test set size than required by the MUMCUT criterion. In such a case, the CUTPNFP criterion will

also be feasible so the NFPs chosen could be chosen to satisfy the CUTPNFP criterion, but this is not necessary. The test size savings also come from not having to generate MNFP tests. Consider $f = ab + cd$. 1101 and 1110 are MUTP tests for term ab . 0111 and 1011 are MUTP tests for term cd . AN NFP of 0101 (for a and c) and an NFP of 1010 (for b and d) completes a MUTP/NFP test set.

MAX-A [46]: Every point from the set of UTPs (for each term) is selected and every point from each set of NFPs (for each literal) is selected.

MAX-B [46]: Every point from the set of UTPs (for each term) is selected and every point from each set of NFPs (for each literal) is selected. In addition, $\log_2(|OTP(S)|)$ (the size of the set of OTPs) and $\log_2(|RFP(S)|)$ (the size of the set of RFPs) are also selected.

MAX-A and MAX-B will only be considered briefly during the comparison of subsumption of tests since they are an excessive extension of the other minimal DNF logic coverage criteria created by Chen and Lau [7].

Minimal DNF Logic Faults

One method for evaluating tests is to determine which faults in Table 2 a test set is guaranteed to detect. These faults are important because based on the competent programmer hypothesis [11] (which states that competent programmers write programs that differ from a correct version by a few simple faults), these faults are likely to occur.

Table 2 Minimal DNF Faults [30]

Fault	Description
Expression Negation Fault (ENF)	Predicate implemented as its negation: $ab + c$ implemented as $\sim(ab + c)$.

Fault	Description
Term Negation Fault (TNF)	A term is negated: $ab + c$ implemented as $\sim(ab) + c$.
Operator Reference Fault + (ORF+)	Replacing OR with AND: $a + b$ implemented as ab .
Operator Reference Fault . (ORF.)	Replacing AND with OR: ab implemented as $a + b$.
Literal Negation Fault (LNF)	A literal is negated: ab implemented as $a\sim b$.
Literal Reference Fault (LRF)	A literal is replaced by a literal or the negation of a literal not in the term: $ab + cd$ implemented as $cb + cd$ or as $\sim cb + cd$.
Term Omission Fault (TOF)	A term is omitted: $ab + cd$ implemented as ab .
Literal Omission Fault (LOF)	A literal is omitted: ab implemented as a .
Literal Insertion Fault (LIF)	A literal not in a term is inserted as itself or as its negation: $ab + cd$ implemented as $abc + cd$ or as $ab\sim c + cd$.

Chen, Lau and Yu [8] established circumstances under which each fault type will be detected. These conditions are highlighted in Table 3. In this table, the TNF was added as a new fault type and MUTP/NFP was added as a new logic coverage criterion by Kaminski, Williams and Ammann [24].

Table 3 Relation Between Faults and Criteria [8, 24]

Fault	Test Needed to Catch Fault	Criteria guaranteed to detect fault if satisfied
Expression Negation Fault (ENF)	Any point in will detect this fault.	Any criterion discussed herein.
Term Negation Fault	If a term p_i in S is implemented as its negation, then any $UTP_i(S)$ or any false point	Any criterion discussed herein.

Fault	Test Needed to Catch Fault	Criteria guaranteed to detect fault if satisfied
(TNF)	(and thus any $NFP_{i,\bar{j}}(S)$) will detect the fault.	
Literal Negation Fault (LNF)	If the literal x^i_j in p_i in S is implemented as its negation, then any $UTP_i(S)$ or $NFP_{i,\bar{j}}(S)$ will reveal the fault.	UTPC, MUTP, MNFP, CUTPNFP, PCUTPNFP, MUMCUT, MUTP/NFP, MAX-A, MAX-B
Term Omission Fault (TOF)	If the term p_i in S is omitted, any $UTP_i(S)$ will detect the fault.	UTPC, MUTP, CUTPNFP, PCUTPNFP, MUMCUT, MUTP/NFP, MAX-A, MAX-B
Operator Reference Fault (ORF)	If OR is replaced by AND (ORF+) between terms p_i and p_{i+1} in S , $UTP_i(S)$ or $UTP_{i+1}(S)$ will detect the fault. If AND is replaced by OR (ORF-) between literals x^i_j and x^i_{j+1} in S , any $NFP_i(S)$ will detect the fault. Also any false point such that either of the two newly created terms evaluates to true will detect the fault.	For OR replaced by AND: UTPC, MUTP, CUTPNFP, PCUTPNFP, MUMCUT, MUTP/NFP, MAX-A, MAX-B For AND replaced by OR: MNFP, CUTPNFP, PCUTPNFP, MUMCUT, MUTP/NFP, MAX-A, MAX-B
Literal Omission Fault (LOF)	If the literal x^i_j in p_i in S is omitted from the term, any $NFP_{i,\bar{j}}(S)$ will detect the fault.	MNFP, CUTPNFP, PCUTPNFP, MUMCUT, MUTP/NFP, MAX-A, MAX-B
Literal Insertion Fault (LIF)	If some literal in S that is not in p_i is implemented in p_i or is implemented as its negation in p_i , then any set of $UTP_i(S)$ where all other literals not in p_i attain the values 0 and 1 will detect the fault. If such a set is infeasible, the LIF results in an equivalent fault.	MUTP, MUMCUT, MUTP/NFP, MAX-A, MAX-B
Literal Reference Fault (LRF)	If the literal x^i_j in p_i in S is implemented as some other literal not in p_i or the negation of some other literal not in p_i , then any of the following will detect the fault: any set of $UTP_i(S)$ where all other literals not in p_i attain the values 0 and 1; any set of $NFP_{i,\bar{j}}(S)$ where all other literals not in p_i attain the values 0 and 1; a pairing of $UTP_i(S)$ and $NFP_{i,\bar{j}}(S)$ that differs only in the value of literal x^i_j . If a pairing of $UTP_i(S)$ and $NFP_{i,\bar{j}}(S)$ that differs only in the value of literal x^i_j is infeasible, the LRF results in an equivalent fault.	MUTP (when feasible), MNFP (when feasible), CUTPNFP (when feasible), MUMCUT, MUTP/NFP (when feasible or when f is a singular Boolean expression when expressed in minimal DNF), MAX-A, MAX-B

The fault hierarchy introduced by Lau and Yu [30] includes the faults in Table 3 and is shown in Figure 2. In this hierarchy, an arrow from a source fault to a destination fault indicates that if a test detects a fault belonging to the source fault class, it will also detect a corresponding fault that belongs to the destination fault class. More specifically, if a test set that satisfies criterion X is guaranteed to detect fault A, where fault A points to fault B, then any test set that satisfies criterion X is guaranteed to detect fault B. The three fault types in the left column in the figure can only be detected by UTPs, the three fault types in the right column in the figure can only be detected by NFPs, and the four fault types in the middle column can all be detected either by UTPs or NFPs. This agrees with the “Test Needed to Catch Fault” column in Table 3. This dissertation extends the fault hierarchy to account for feasibility in Figure 6 in section 4.3 and further extends the fault hierarchy to include faults from several mutation operators in Figure 9 in section 7.1.

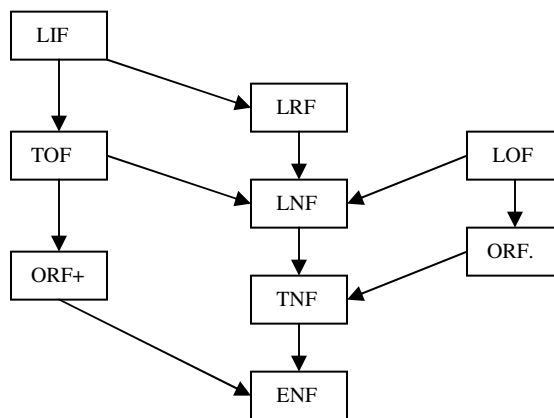


Figure 2 Lau and Yu’s Fault Hierarchy [30]

Subsumption

Subsumption is one method for comparing coverage criteria. If any test set that satisfies criterion X also satisfies criterion Y, then X subsumes Y. Although a test set that satisfies criterion X will satisfy criterion Y, a test set that satisfies criterion Y may have tests that are not in a test set that satisfies criterion X. Consequently, tests that satisfy criterion Y may catch faults that tests that satisfy criterion X miss. Figure 3 summarizes the subsumption relations among criteria. An arrow indicates that the source criterion subsumes the destination criterion. RICC and GICC are in their own hierarchy because the subsumption relations between them and the other criteria are not known.

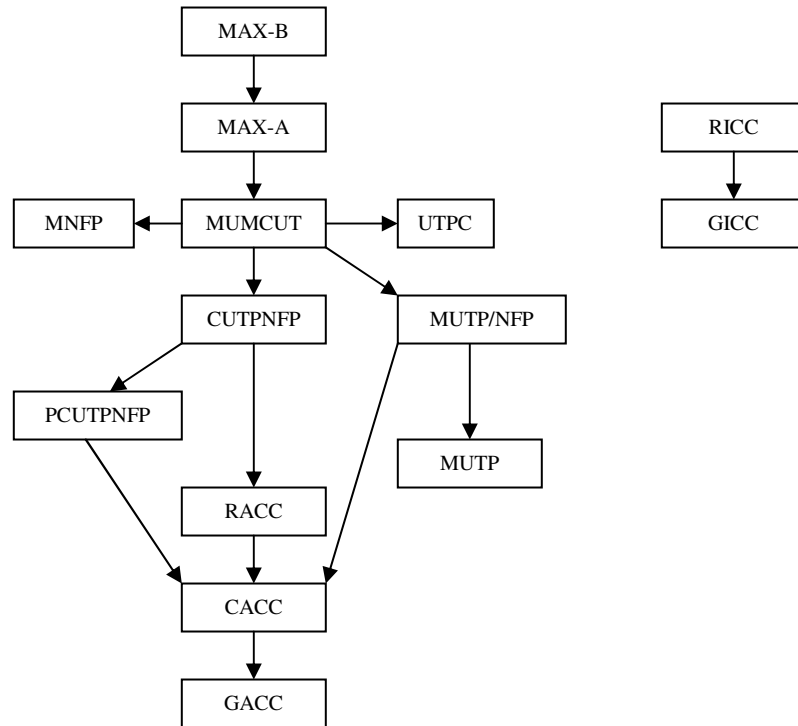


Figure 3 Subsumption Hierarchy [24]

Summary

Table 4 gives a summary of test type, fault detection, subsumption relation and test set size for each of the criteria presented.

Table 4 Logic Coverage Criteria Summary [24]

Test Name	Test Type	Guaranteed Faults Detected	Subsumes	Subsumed by	Minimum Test Size	Maximum Test Size
Restricted Active Clause Coverage (RACC)	Semantic	ENF, TNF	CACC, GACC	CUTPNFP, MUMCUT, MAX-A, MAX-B	$n+1$, where n is the number of unique literals in function f .	$2n$, where n is the number of unique literals in function f .
Correlated Active Clause Coverage (CACC)	Semantic	ENF, TNF	GACC	RACC, MUTP/NFP, PCUTPNFP, CUTPNFP, MUMCUT, MAX-A, MAX-B	$RUTW(2*\sqrt{n})$, where $RUTW$ = "Round up to the nearest whole number" and n is the number of unique literals	$2n$, where n is the number of unique literals in function f .
General Active Clause Coverage (GACC)	Semantic	ENF, TNF	-	CACC, RACC, MUTP/NFP, CUTPNFP, MUMCUT, MAX-A, MAX-B	2, if Boolean Derivative evaluates to 1 for all literals, up to $RUTW(2*\sqrt{n})$, since $CACC=GACC$ for many Boolean functions, where n is the number of unique literals in function f .	$2n$, where n is the number of unique literals in function f .
Restricted Inactive Clause Coverage (RICC)	Semantic	ENF, TNF	GICC	-	$2n$, where n is the number of unique literals in function f . (true when all RICC tests are feasible)	$4n$, where n is the number of unique literals in function f .
General Inactive Clause Coverage (GICC)	Semantic	ENF, TNF	-	RICC	$2n$, where n is the number of unique literals in function f . (true when all GICC tests are feasible)	$4n$, where n is the number of unique literals in function f .
Unique True Point Coverage (UTPC)	Syntactic	ENF, TNF, LNF, TOF, ORF (when OR is replaced with AND)	-	MUMCUT, MAX-A, MAX-B	$n+1$ where n is the number of literals in Boolean function f .	$m + \prod_{i=1}^k n_i$, where m is the number of terms in Boolean function f and n_i is the number of literals in term i .
Multiple Unique True Point (MUTP)*	Syntactic	ENF, TNF, LNF, TOF, ORF (when OR is replaced	-	MUTP/NFP, MUMCUT, MAX-A, MAX-B	m to $2m$, where m is the number of terms in Boolean function f .	$2m(n-1)$, where m is the number of terms in function f and n is the

Test Name	Test Type	Guaranteed Faults Detected	Subsumes	Subsumed by	Minimum Test Size	Maximum Test Size
		with AND), LIF				number of literals in function f .
Multiple Near False Point (MNFP)*	Syntactic	ENF, TNF, LNF, ORF (when AND is replaced with OR), LOF	-	MUMCUT, MAX-A, MAX-B	1 (when infeasibilities arise); Uncertain otherwise.	$\frac{mn^2}{2}$, where m is the number of terms in function f and n is the number of literals in function f .
Corresponding Unique True Point Next False Point (CUTPNFP)*	Syntactic	ENF, TNF, LNF, TOF, ORF, LOF	RACC, CACC, GACC, PCUTPNFP	MUMCUT, MAX-A, MAX-B	$\sum_{i=1}^m n_i + 1$, where n_i is the number of literals in term i and m the number of terms in f .	$2mn$, where m is the number of terms in function f and n is the number of literals in function f .
Partial Corresponding Unique True Point Near False Point (PCUTPNFP)**	Syntactic	ENF, TNF, LNF, TOF, ORF., ORF+, LOF	CACC, GACC	CUTPNFP, MUMCUT, MAX-A, MAX-B	$\sum_{i=1}^m n_i + 1$ where n_i is number of literals in term i and m is number of terms	Uncertain, but less than $2mn$ where m is the number of terms and n is the number of literals
MUTP / MNFP / CUTPNFP Strategy (MUMCUT)	Syntactic	ENF, TNF, LNF, TOF, ORF, LOF, LIF, LRF	RACC, CACC, GACC, MUTP, MUTP/NFP, MNFP, UTPC, CUTPNFP, PCUTPNFP	MAX-A, MAX-B	When infeasibilities arise – m to $2m + 1$ (where m is the number of terms in Boolean function f). Uncertain otherwise.	$2m(n-1) + \frac{mn^2}{2}$, where m is the number of terms in function f and n is the number of literals in function f .
Multiple Unique True Point / Near False Point (MUTP / NFP) ***	Syntactic	ENF, TNF, LNF, TOF, ORF, LOF, LIF	CACC, GACC, MUTP	MUMCUT, MAX-A, MAX-B	$m + 1$ to $2m + 1$ (where m is the number of terms in Boolean function f).	$2m(n-1) + n$, where m is the number of terms in function f and n is the number of literals in function f .

* The MUTP, MNFP and CUTPNFP criterion are each guaranteed to detect the LRF when feasible.

** The PCUTPNFP criterion, when feasible, is guaranteed to detect any LRF that the MUTP criterion does not detect.

*** The MUTP/ NFP criterion is guaranteed to detect the LRF when feasible or when f is a singular Boolean expression when expressed in minimal DNF.

2.3 Mutation Testing Related Work

This section describes related work in mutation testing. First, a discussion of strong vs. weak mutation is presented. Next, related work on semantic vs. syntactic fault size is discussed. This is followed by a summary of related work on detecting equivalent

mutants. Next, the internal variable problem is introduced and prior research on higher order mutants is presented. The section concludes with an examination of related work on mutation testing for databases and queries.

Mutation testing exists in two forms: weak and strong. In strong mutation testing, a mutant is considered killed if and only if the output of the mutant and original program differ. In weak mutation testing, this is relaxed so that a mutant is considered killed if and only if the program state of the mutant and original program differ after execution of the mutated statement. This research focuses on weak mutation testing. For a logic mutant to be weakly killed, the mutated predicate must be reached and evaluate to a different truth value in the mutant than in the original program. If a mutant is weakly killed, the mutant and original program can have identical outputs. In practice, Offutt and Lee [35] found that weak mutation testing is almost as effective as strong mutation testing, with major computational savings. Thus, most mutants that are weakly killed are strongly killed.

For this research, classifying a logic mutant as not equivalent means it is not equivalent based on weak mutation testing, which almost always means it is not equivalent based on strong mutation testing. When referring to killing logic mutants, the author implies weakly killing the mutants. This is because logic coverage criteria are based on weakly killing mutants in that these criteria examine a particular predicate in isolation. When referring to killing mutants in general, the author implies strongly killing mutants. Thus, classifying a logic mutant as equivalent means that it is equivalent based on weak mutation testing, which means that it will always be equivalent based on strong mutation testing. However, classifying a general mutant as equivalent means it is

equivalent based on strong mutation testing, which almost always means it is equivalent based on weak mutation testing.

Offutt and Hayes [34] introduced the concepts of semantic and syntactic fault models. A syntactic characterization involves the actual code changes that differentiate a faulty and correct program. A semantic characterization views a faulty program as producing incorrect output for one or more inputs. Syntactic fault size is defined as the fewest number of tokens that need to change to produce a correct program. Semantic fault size is defined as the relative size of the input domain for which the output is incorrect. Offutt and Hayes argue that mutants with semantically small faults are desirable because testers must select one of the few inputs that kill them. They also argue that mutants with syntactically small but semantically large faults add little value because most inputs kill them. Offutt, Rothermel and Zapf [37] examined using selective mutation operators to reduce mutant set size. Selective mutation uses operators that produce fewer mutants by experimentally selecting operators that overlap with others. They found that selective mutation provided almost the same coverage as non-selective mutation with much fewer mutants.

Prior methods for automatically detecting equivalent mutants include compiler optimization techniques introduced by Offutt and Craft [33] and constraint-based techniques introduced by Offutt and Pan [36]. Offutt and Pan [36] showed that using constraints was superior to compiler optimization. The best technique for detecting equivalent mutants is program slicing as introduced by Hierons, Harman and Danicic [15]. Program slicing uses decomposition to extract from program statements information

relevant to a particular computation. A slice helps determine what program statements affect the computation of a variable, which can be used to automatically detect some equivalent mutants. Detecting equivalent mutants is formally undecidable, but exclusive-OR algorithms can detect all equivalent *logic* mutants (assuming reachability, propagation and a complete Boolean space). (Complete Boolean space means that it is possible to assign any combination of values to the literals in a predicate. In other words, all points are feasible.) Kaminski et al. [23] developed a technique to detect equivalent logic mutants when a complete Boolean space does not exist.

Finding inputs to kill non-equivalent mutants is formally undecidable and is beyond the scope of this research. However, this problem does become more complex for selective logic mutation because it may not be possible to find an input to kill a mutant that if it could be killed, would guarantee killing other mutants. To address this complexity, Kaminski et al. [23] developed a mutation tool that can aid the tester in finding inputs to kill logic mutants by informing the tester what the values of the variables in the mutated predicate need to be in order to weakly kill the mutant.

Polo, Piattini and Garcia-Rodriguez [39] examined decreasing the cost of mutation testing with second-order mutants. They proposed that the number of mutants can be reduced by half by means of combining the original set of mutants to obtain a new set of mutants, each one with two faults. Jia and Harman [17] show that certain higher order mutants are subsuming in that any test input that kills a subsuming higher order mutant guarantees killing each single order mutant that the higher order mutant is composed of. They describe search based algorithms to identify subsuming higher order mutants and

eliminate non-subsuming higher order mutants, which reduces the number of mutants generated. However, neither Polo et al. nor Jia and Harman focus on logic mutants.

Many researchers have used constraint solvers to populate a test database for the purpose of testing SQL database queries. The most relevant research using constraint solvers is based on work by Emmi et al. [12] and Willmor and Embury [47]. However, these approaches do not focus on an explicit adequacy criterion or fault hierarchy like the TRF-TIF approach. Thus, these approaches do not guarantee detection of a specified set of faults. Kapfhammer and Soffa [25] apply data-flow criteria but at a higher level than that for individual clauses in a query. Chays et al. [4, 5] proposed the AGENDA tool, which focuses on testing at the transaction level, but this is at a higher level than that for individual clauses in a query and individual records in a test database. Suarez-Cabal and Tuya [40] examined multiple condition coverage (MCC) for SELECTS and JOINS in queries by creating a set of coverage trees. Their research focused on the level of individual records. However, the limitation is that the coverage trees grow exponentially. Chan and Cheung [3] transformed a SQL query into a procedural language to which various criteria can be applied. However, this approach shares some of the same drawbacks mentioned above as well as problems dealing with preserving query semantics during translation. To overcome these limitations, Tuya et al. [42] developed an approach called Full Predicate Coverage (SQLFpc), which is based on Masking MCDC or CACC. Their approach avoids the problem of exponential growth.

Halfond and Orso [13] presented an approach known as “command form coverage”, which focuses on how the String object containing the actual SQL query in a program is

constructed. In other words, they examine how different variations of a query can be formed dynamically at a single point in the source code. Coverage is measured as the ratio of different queries actually formed by the inputs to the program to the total number of different queries that are possible. This approach is complementary to Tuya’s approach (and the TRF-TIF approach) because the “command form coverage” approach does not consider logic coverage of the queries themselves.

2.4 Mutation Testing Background Material

In this dissertation, TRF-TIF logic mutation is compared with three other mutation tools. These are typical logic mutation, muJava and SQLMutation. Typical logic mutation refers to a hypothetical tool including a common set of mutation operators, most of which are described by Ammann and Offutt [2]. A subset of these mutation operators are in a mutation testing tool called muJava developed by Ma et al. [31]. muJava generates both logic and non-logic mutants. It is used as a mutation tool for software and also as a fault seeder to introduce faults into software. Tuya et al. [44] have built a query mutation tool known as the SQLMutation tool. It allows mutants to be generated interactively from a Web browser. The mutation operators in this tool cover a wide range of SQL syntax and semantics as described by Tuya et al. [42]. SQLFpc is based on masking MCDC (CACC). It can be used to seed faults into queries.

3 Thesis Contributions

This research evaluates a new syntactic logic coverage criterion called Minimal-MUMCUT and a new logic mutation approach called TRF-TIF logic mutation. The high level hypothesis that is evaluated has two parts.

HIGH LEVEL HYPOTHESIS PART I:

The Minimal-MUMCUT logic coverage criterion provides a way to reduce test set size and/or improve logic fault detection when compared to current logic coverage criteria.

HIGH LEVEL HYPOTHESIS PART II:

The TRF-TIF logic mutation approach provides a way to reduce mutant set size (and equivalent mutant set size) and/or improve logic fault detection when compared to current mutation approaches.

The evaluation of the high level hypothesis leads to several contributions that relate to Figure 1 in section 1.4. Each contribution is labeled with a number and a letter. The number indicates which logic coverage criterion Minimal-MUMCUT is being compared with or which mutation approach TRF-TIF logic mutation is being compared with. The letter indicates the feature that is being compared.

The numbering is as follows:

- 1 – Comparison of Minimal-MUMCUT [22] with MUMCUT [50]
- 2 – Comparison of Minimal-MUMCUT with ACC/ICC [2]
- 3 - Comparison of TRF-TIF logic mutation [18] with Typical logic mutation [2]
- 4 - Comparison of TRF-TIF logic mutation with muJava [31]
- 5 - Comparison of TRF-TIF logic mutation with SQLMutation [44]

The lettering is as follows:

- a – Test set size or mutant set size
- b – Equivalent mutant set size
- c – Single Minimal DNF logic fault detection (innermost fault oval in Figure 1)
- d – Double Minimal DNF logic fault detection (innermost fault oval in Figure 1)
- e – Single logic fault detection regardless of format (center fault oval in Figure 1)
- f – General fault detection (outermost fault oval in Figure 1)

Letter “f” is a special case in that no comparison between criteria or mutation approaches above is made. Instead letter “f” indicates how well tests that weakly kill all TRF-TIF mutants (and hence satisfy the Minimal-MUMCUT criterion) detect faults in general (both non-logic faults and logic faults without regards to predicate format). Since muJava is used to seed the general software faults, letter “f” is paired with number 4. Since SQLMutation is used to seed the general query faults, letter “f” is also paired with number 5.

Some pairings between number and letter are not applicable. MUMCUT and ACC/ICC are criteria for which equivalent mutant set size does not apply and thus there is no contribution 1b or 2b. Of MUMCUT, ACC/ICC, typical logic mutation, muJava and SQLMutation, only MUMCUT behaves differently depending on predicate format so there is no contribution 2e, 3e, 4e, or 5e. Table 5 displays a summary of the contributions.

Table 5 Contribution Summary

	Test / Mutant Set Size	Equiv Mutant Set Size	Single DNF Fault Detection	Double DNF Fault Detection	General Logic Fault Detection	General Fault Detection
MUMCUT (Chapter 5)	1a [22]	N/A	1c [19]	1d [20]	1e [22]	N/A
ACC/ICC (Chapter 6)	2a [21]	N/A	2c [24]	2d [21]	N/A	N/A
Typical logic mutation (Chapter 8)	3a [18]	3b [18]	3c [18]	3d [18]	N/A	N/A
muJava (Chapter 9)	4a [23]	4b [23]	4c [23]	4d [23]	N/A	4f [23]
SQLMutation (Chapter 10)	5a [23]	5b [23]	5c [23]	5d [23]	N/A	5f [23]

The following subsections describe the contributions listed in Table 5 beginning with the top row and working downwards. That is, section 3.1 corresponds to the first row in Table 5, section 3.2 corresponds to the second row in Table 5 and so on. Each subsection gives the results of the contribution and how the contribution was obtained (the theoretical or empirical method used). In the case of empirical contributions, the results are given for the particular empirical subjects chosen. A general discussion of threats to validity appears at the end of this chapter. More detailed discussions on the empirical

studies, including more specific threats to validity where appropriate, appear in chapters 5, 6 and 8-10.

The contributions focus on reducing test set and mutant set size as well as increasing fault detection. In applications where testing is extremely expensive, reducing test set size or mutant set size by one can be valuable. In applications where testing is inexpensive, a large reduction in test set size or mutant set size may make little difference. In safety-critical applications, missing detection of even a single fault can have drastic consequences. Conversely, in other types of applications, missing detection of many faults may have little impact on the user. To provide a uniform basis of evaluation, four levels of reduction in test/mutant set size (Table 6) and four levels of reduction in fault detection capability (Table 7) are used. The results are evaluated in the context of these levels. The goal is to *decrease* test set size or mutant set size and to *increase* fault detection capability.

Table 6 Terms for Reduction in Test Set Size or Mutant Set Size

Term	Ratio of Test Set Size or Mutant Set Size to Size of Interest
Unsubstantial	(75%, 100%)
Substantial	(50%, 75%]
Significant	(10%, 50%]
Very Significant	(0%, 10%]

The term *unsubstantial* will be used when test set size or mutant set size is reduced to between 75% (exclusive) and 100% (exclusive) of the size of interest, the term *substantial* will be used when test set size or mutant set size is reduced to between 50%

(exclusive) and 75% (inclusive) of the size of interest, the term *significant* will be used when test set size or mutant set size is reduced to between 10% (exclusive) and 50% (inclusive) of the size of interest and the term *very significant* will be used when test set size or mutant set size is reduced to between 0% (exclusive) and 10% (inclusive) of the size of interest.

Table 7 Terms for Reduction in Fault Detection Capability

Term	Ratio of Number of Faults Detected to Number of Faults of Interest
Small Minority	[0%, 25%)
Minority	[25%, 50%)
Majority	[50%, 90%)
Vast Majority	[90%, 100%)

The term *small minority* will be used when fault detection capability is between 0% (inclusive) and 25% (exclusive) of all faults of interest. The term *minority* will be used when fault detection capability is between 25% (inclusive) and 50% (exclusive) of all faults of interest, the term *majority* will be used when fault detection capability is between 50% (inclusive) and 90% (exclusive) of all faults of interest and the term *vast majority* will be used when fault detection capability is between 90% (inclusive) and 100% (exclusive) of all faults of interest.

3.1 Contributions Comparing Minimal-MUMCUT with MUMCUT

Contribution 1a Part I:

For a sample of minimal DNF predicates with 5 or more unique literals, Minimal-MUMCUT test set size is *substantially* less than MUMCUT test set size.

Empirical result: Minimal-MUMCUT test set size was 64% of MUMCUT test set size.

Contribution 1a Part II:

For a sample of minimal DNF predicates with 5 or more unique literals, Union Minimal-MUMCUT test set size is *very significantly* less than MUMCUT extension test set size.

Empirical result: Union Minimal-MUMCUT test set size was 3% of MUMCUT extension test set size.

Contribution 1c:

For minimal DNF predicates, Minimal-MUMCUT tests and MUMCUT tests have the same guaranteed single logic fault detection (9 of 9 fault types in Lau and Yu's fault hierarchy).

Contribution 1d Part I:

For minimal DNF predicates, Minimal-MUMCUT and MUMCUT tests have the same guaranteed double logic fault detection (84 of 92 double fault types in Lau and Yu's fault hierarchy).

Contribution 1d Part II:

For a sample of minimal DNF predicates with 5 or more unique literals, Minimal-MUMCUT tests and MUMCUT tests actually detect the *vast majority* of the double logic faults that are not guaranteed to be detected.

Empirical result: Minimal-MUMCUT tests and MUMCUT tests detected 99% of the double logic faults that are not guaranteed to be detected.

Contribution 1e Part I:

For the *majority* of predicates with at least 3 unique literals in a sample from avionics software, fault detection provided by Union Minimal-MUMCUT tests and MUMCUT tests is not compromised due to predicate format because the predicates are in minimal DNF, minimal CNF, or both.

Empirical result: For 85% of the predicates, fault detection was not compromised for either Union Minimal-MUMCUT tests or MUMCUT tests.

Contribution 1e Part II:

Union Minimal-MUMCUT tests and MUMCUT tests detect the *vast majority* of logic faults for a sample of predicates with 5 or more unique literals when the minimal DNF/CNF assumption does not hold.

Empirical result: Union Minimal-MUMCUT tests and MUMCUT tests detected 98% of the logic faults.

These contributions establish that the Minimal-MUMCUT criterion is more efficient than the MUMCUT criterion in that it reduces MUMCUT test set size while preserving MUMCUT logic fault detection for minimal DNF/CNF predicates. Similarly, these contributions establish that the Union Minimal-MUMCUT criterion is more efficient than the MUMCUT extension criterion in that it reduces MUMCUT extension test set size with minimal impact on logic fault detection for general form predicates.

For contribution 1a Part I, Minimal-MUMCUT and MUMCUT test set size are compared for 19 predicates in avionics software. For contribution 1a Part II, Union Minimal-MUMCUT and MUMCUT extension test set size are compared for 10 general form predicates in avionics software. For contributions 1c and 1d Part I, proofs are given that Minimal-MUMCUT tests detect the same single and double fault types in Lau and Yu's fault hierarchy as MUMCUT tests. For contribution 1d Part II, a proof is given relating criterion feasibility to the conditions under which double fault types can go undetected. For any double fault type that is not guaranteed to be detected, an empirical study is conducted in which all possible double faults that correspond to the double fault type are seeded into 19 predicates. It is then determined what percentage of these faults go undetected by Minimal-MUMCUT tests. For contribution 1e Part I, 20,256 predicates in avionics software are examined to determine the percentage in minimal CNF, minimal DNF, neither, or both. For contribution 1e Part II, 3570 non-equivalent faults are seeded into 10 general form predicates and for each fault, it is determined if a Union Minimal-MUMCUT test set detects it.

3.2 Contributions Comparing Minimal-MUMCUT with RACC and RICC

Contribution 2a:

For a sample of minimal DNF predicates with 5 or more unique literals, RACC test set size is *significantly* smaller than Minimal-MUMCUT test set size.

Empirical result: RACC test set size was 25% of Minimal-MUMCUT test set size.

Contribution 2c Part I:

For minimal DNF predicates, a test set that satisfies either RACC or RICC guarantees detection of 2 of the 9 single fault types in Lau and Yu's fault hierarchy (ENF and TNF).

Contribution 2c Part II:

For a sample of minimal DNF predicates with 5 or more unique literals, RACC tests detect a *minority* of the minimal DNF single faults they are not guaranteed to detect.

Empirical result: RACC tests detected 34% of all minimal DNF single faults they are not guaranteed to detect.

Contribution 2d:

For minimal DNF predicates, a test set that satisfies either RACC or RICC guarantees detecting 22 of the 92 double fault types in Lau and Yu's fault hierarchy.

These contributions establish that although RACC/RICC test set size is less than Minimal-MUMCUT test set size, Minimal-MUMCUT tests detect more minimal DNF logic faults than RACC/RICC tests.

For contribution 2a, RACC test set size and Minimal-MUMCUT test set size are compared for 19 predicates in avionics software. For contribution 2c Part I, RACC and RICC tests are created for some small predicates. Then faults are introduced into the predicates corresponding to the fault types in Lau and Yu's fault hierarchy and it is determined which faults are detected by the tests. For contribution 2c Part II, a RACC test set is created for each of the 19 predicates mentioned above and all of the faults in Lau and Yu's fault hierarchy are generated. Then, for each predicate, it is determined whether or not the tests detect each fault. For contribution 2d, the detection conditions for all double fault types in Lau and Yu's fault hierarchy are analyzed in terms of which are guaranteed to be satisfied by a RACC test set and which are guaranteed to be satisfied by a RICC test set.

3.3 Contributions Comparing TRF-TIF Logic Mutation with Typical Logic Mutation

Contribution 3a:

For software containing a sample of minimal DNF predicates with 5 or more unique literals, TRF-TIF mutant set size is *very significantly* smaller than typical logic mutant set size (where typical refers to a set of mutants that would be produced by a set of common mutation operators, most of which are specified by Ammann and Offutt [2]).

Empirical result: 6%.

Contribution 3b:

For software with minimal DNF predicates, a TRF-TIF mutant set contains the same number as or fewer weakly equivalent mutants than a typical logic mutant set (assuming any infeasible combinations of values of unique literals are specified).

Contribution 3c Part I:

For software containing minimal DNF predicates, tests weakly killing all TRF-TIF mutants are guaranteed to detect all 9 single fault types in Lau and Yu's fault hierarchy while tests weakly killing all typical logic mutants are guaranteed to detect 7 of the 9 single fault types in Lau and Yu's fault hierarchy.

Contribution 3c Part II:

For software containing a sample of minimal DNF predicates with 5 or more unique literals, tests weakly killing all typical logic mutants detect the *majority* of the single faults in Kaminski and Ammann's [18] extension to Lau and Yu's fault hierarchy (as opposed to the 100% detection percentage by tests weakly killing all TRF-TIF mutants).

Empirical result: Tests weakly killing all typical logic mutants detected 75% of the single faults.

Contribution 3d Part I:

For software containing minimal DNF predicates, tests that weakly kill all TRF-TIF mutants and tests that weakly kill all typical logic mutants both are guaranteed to detect 84 of the 92 double fault types in Lau and Yu's fault hierarchy.

Contribution 3d Part II:

For the vast majority of minimal DNF predicates in software containing a sample of minimal DNF predicates with 5 or more unique literals, tests that weakly kill all TRF-TIF mutants will guarantee detection of 91 of the 92 double fault types in Lau and Yu's fault hierarchy but tests that weakly kill all typical logic mutants will actually detect only 84 of the 92 double fault types.

Empirical result: For 100% of the predicates, this was the case.

These contributions establish that using TRF-TIF logic mutation instead of typical logic mutation reduces mutant set size and equivalent mutant set size while at the same time guaranteeing more minimal DNF/CNF logic faults are detected. Typical logic mutation refers to a mutation approach that includes a common set of logic mutation operators, most of which are specified by Ammann and Offutt [2].

For contribution 3a, TRF-TIF mutant set size and typical logic mutant set size for a program containing 19 minimal DNF predicates is compared. For contribution 3b, the mutation operators in TRF-TIF logic mutation and typical logic mutation are analyzed in terms of producing equivalent faults. It is shown that TRF-TIF logic mutation can

automatically detect weakly equivalent logic mutants. For contribution 3c Part I, tests that kill all typical logic mutants are theoretically analyzed in terms of which fault types they guarantee detecting in Lau and Yu's fault hierarchy. For contribution 3c Part II, all faults in Kaminski and Ammann's [18] expanded fault hierarchy are seeded into 19 predicates. It is shown how many faults are detected by a test that weakly kills all TRF-TIF mutants and by a test that weakly kills all typical logic mutants. For contribution 3d Part I, tests killing all typical logic mutants are theoretically analyzed in terms of which double fault types they guarantee detecting in Lau and Yu's fault hierarchy. For contribution 3d Part II, it is determined which double faults will be detected by tests weakly killing all TRF-TIF mutants for a program with 19 predicates. For the same program, it is shown which double faults are actually detected by tests weakly killing all typical logic mutants.

3.4 Contributions comparing TRF-TIF Logic Mutation with muJava

Contribution 4a Part I:

For a sample of Java programs having a Unique Literals Ratio greater than 0.10, TRF-TIF mutant set size is *significantly less* than muJava general mutant set size (the mutant set size of all muJava mutants).

Empirical result: TRF-TIF mutant set size was 14% of muJava general mutant set size.

Contribution 4a Part II:

For a sample of minimal DNF predicates with 5 or more unique literals, TRF-TIF mutant set size is *significantly less* than muJava logic mutant set size (the mutant set size of all muJava logic mutants).

Empirical result: TRF-TIF mutant set size was 25% of muJava logic mutant set size.

Contribution 4b:

For a sample of Java programs having a Unique Literals Ratio greater than 0.10, strongly equivalent TRF-TIF mutant set size is *significantly less* than strongly equivalent muJava mutant set size.

Empirical result: Strongly equivalent TRF-TIF mutant set size was 13% of strongly equivalent muJava mutant set size.

Contribution 4c:

For software containing minimal DNF predicates, tests weakly killing all muJava mutants are guaranteed to detect 5 of the 9 single fault types in Lau and Yu's fault hierarchy.

Contribution 4d:

For software with minimal DNF predicates, tests weakly killing all muJava mutants are guaranteed to detect fewer double fault types in Lau and Yu's fault hierarchy than Minimal-MUMCUT tests.

Contribution 4f:

For a sample of Java programs having a Unique Literals Ratio greater than 0.10, tests weakly killing all TRF-TIF mutants strongly kill the *vast majority* of strongly non-equivalent muJava mutants.

Empirical result: Tests weakly killing all TRF-TIF mutants strongly killed 90% of strongly non-equivalent muJava mutants.

These contributions establish that using TRF-TIF logic mutation instead of muJava reduces mutant set size and equivalent mutant set size while at the same time guaranteeing more minimal DNF/CNF logic faults are detected. Also, they establish that tests that weakly kill all TRF-TIF mutants detect a high percentage of non-logic software faults.

For contribution 4a Part I, TRF-TIF mutant set size and muJava general mutant set size are evaluated for 30 small Java programs and 1 larger Open Source Software Java program. MuJava general mutant set size is the number of all muJava mutants (both logic and non-logic). For contribution 4a Part II, TRF-TIF mutant set size and muJava logic mutant set size for 19 minimal DNF predicates are compared. For contribution 4b, the number of strongly equivalent mutants in the TRF-TIF mutant set and muJava mutant set are determined manually for each of the small programs. For contributions 4c and 4d, proofs are given showing what single and double fault types in Lau and Yu's hierarchy such a muJava test set is guaranteed to detect. For contribution 4f, tests that weakly kill all TRF-TIF mutants are created for each program used in contribution 4a Part I. For each

small program, it is determined what percentage of the strongly non-equivalent muJava mutants are strongly killed by the tests. For the larger program, it is determined what percentage of the muJava mutants are weakly killed by the tests, assuming 10% of the muJava mutants are weakly equivalent [23].

3.5 Contributions comparing TRF-TIF Logic Mutation with SQLMutation

Contribution 5a:

For a sample of queries having minimal DNF WHERE clauses with 3 or more unique literals, TRF-TIF mutant set size is *very significantly* less than SQLMutation mutant set size.

Empirical result: TRF-TIF mutant set size was 2% of SQLMutation mutant set size.

Contribution 5b:

For queries having minimal DNF WHERE clauses with 3 or more unique literals, equivalent TRF-TIF mutant set size is *very significantly* smaller than SQLMutation mutant set size assuming a complete Boolean space.

Empirical result: TRF-TIF mutant set size was 0% of SQLMutation mutant set size.

Contribution 5c:

For queries having minimal DNF WHERE clauses, tests weakly killing all SQLMutation mutants are guaranteed to detect 2 of the 9 single fault types in Lau and Yu's fault hierarchy.

Contribution 5d:

For queries having minimal DNF WHERE clauses, tests weakly killing all SQLMutation mutants are guaranteed to detect 22 of the 92 double fault types in Lau and Yu's fault hierarchy.

Contribution 5f Part I:

For a sample of queries having minimal DNF WHERE clauses with 3 or more unique literals, tests killing all TRF-TIF mutants kill the *vast majority* of non-equivalent SQLMutation mutants.

Empirical result: Tests killing all TRF-TIF mutants killed 90% of non-equivalent SQLMutation mutants.

Contribution 5f Part II:

For a sample of queries having minimal DNF WHERE clauses with 3 or more unique literals, tests killing all TRF-TIF mutants kill 10 times as many SQLMutation mutants as a randomly generated test set of the same size.

Empirical result: Tests killing all TRF-TIF mutants killed 20 times as many SQLMutation mutants as a test set generated randomly of the same size.

These contributions establish that using TRF-TIF logic mutation instead of SQLMutation reduces mutant set size and equivalent mutant set size while at the same time guaranteeing more minimal DNF/CNF logic faults are detected. Also, they establish

that tests that weakly kill all TRF-TIF mutants detect a high percentage of non-logic query faults.

For contribution 5a, TRF-TIF mutant set size and SQLMutation mutant set size are evaluated for 10 queries from an open source project. Each query has at least 4 unique literals in its minimal DNF WHERE clause. For contribution 5b, the number of equivalent mutants in the TRF-TIF mutant set is determined manually and the number of equivalent mutants in the SQLMutation mutant is estimated (as given by the SQLMutation tool author). Results of contributions 2c Part I and 2d are used to confirm contributions 5c and 5d respectively, as SQLMutation is based on CACC. For contribution 5f Part I, tests that weakly kill all TRF-TIF mutants are created for each query used in Contribution 5a. For each query, it is determined what percentage of the non-equivalent SQLMutation mutants are killed by the tests, assuming that 6% to 8% of the SQLMutation mutants are equivalent [23]. For contribution 5f Part II, a random test set generated by Tuya et al. [42] is selected for 6 of the 10 queries [42]. For each query, it is determined what percentage of the SQLMutation mutants are killed by the random test sets.

3.6 General Threats to Validity

A general threat to validity for all of the empirical studies is that it cannot be claimed that the predicates, software and queries selected are representative samples from a population. Also, sample size was fairly small. Thus, formal claims of significance cannot be made. For example, much of the empirical research regarding the logic

coverage criteria comparisons was performed on a sample of 19 predicates extracted from avionics traffic collision avoidance software. For software mutation testing, empirical results were based primarily on a sample of utility methods extracted from the Java API. For query mutation testing, empirical results were based on a sample of 10 queries extracted from an Open Source project. A positive aspect is that if future research performed by others using different predicates, software and queries obtains similar results, the conclusions are strengthened.

The rest of the dissertation is organized as follows. Chapter 4 provides an overview of the Minimal-MUMCUT criterion. Chapter 5 presents results comparing the Minimal-MUMCUT criterion with the MUMCUT criterion. Chapter 5 corresponds to row 1 in Table 5 and each section in Chapter 5 corresponds to a column in Table 5. Chapter 6 presents results comparing the Minimal-MUMCUT criterion with the RACC and RICC criteria. Chapter 6 corresponds to row 2 in Table 5 and each section in Chapter 6 corresponds to a column in Table 5. Chapter 7 provides an overview of TRF-TIF logic mutation. Chapter 8 presents results comparing TRF-TIF logic mutation with typical logic mutation. Chapter 8 corresponds to row 3 in Table 5 and each section in Chapter 8 corresponds to a column in Table 5. Chapter 9 presents results comparing TRF-TIF logic mutation with muJava. Chapter 9 corresponds to row 4 in Table 5 and each section in Chapter 9 corresponds to a column in Table 5. Chapter 10 presents results comparing TRF-TIF logic mutation with SQLMutation. Chapter 10 corresponds to row 5 in Table 5 and each section in Chapter 10 corresponds to a column in Table 5. Chapter 11 discusses conclusions.

4 The Minimal-MUMCUT Criterion

Logic coverage criteria exist that require small test set size, but they do not guarantee detection of common logic faults. Conversely, logic coverage criteria exist that guarantee detection of common logic faults, but these criteria require a large test set size. Part of the reason for this is that current logic coverage criteria (such as the MUMCUT criterion) do not handle infeasibility efficiently, which in turn results in unnecessary tests in that all faults in Lau and Yu's fault hierarchy can still be detected even when one or more tests are removed. The Minimal-MUMCUT criterion improves on the MUMCUT criterion by using feasibility analysis to remove tests yet still guarantee fault detection.

The term *Minimal* in Minimal-MUMCUT is used to refer to the fact that if any test in a Minimal-MUMCUT test set is removed, fault detection is sacrificed for the fault types in Lau and Yu's fault hierarchy. This is different than the term *minimized* which implies that the test set size is as small as possible. Thus, for the Minimal-MUMCUT test sets created, it may be possible to construct a smaller test set that satisfies Minimal-MUMCUT. However, it is guaranteed that if even a single test is removed from a Minimal-MUMCUT test set, fault detection will be sacrificed for the fault types in Lau and Yu's fault hierarchy.

The rest of this chapter is organized as follows. The remainder of section 4.1 describes the algorithm used to generate a Minimal-MUMCUT test set given a minimal

DNF predicate and then updates the subsumption hierarchy in Figure 3 in section 2.2 and the logic coverage criteria summary in Table 4 in section 2.2 to include the Minimal-MUMCUT criterion. Section 4.2 discusses Minimal-MUMCUT test set size and the close relation between test set size, feasibility and LRF detection. Sections 4.3 and 4.4 describe single and double minimal DNF fault detection of the Minimal-MUMCUT criterion, respectively. Section 4.5 describes general logic fault detection of the Minimal-MUMCUT criterion when the minimal DNF assumption fails to hold.

4.1 Overview of the Minimal-MUMCUT Criterion

Figure 4 below gives a visual description of the algorithm used to build a Minimal-MUMCUT test set for minimal DNF predicates. The dashed arrows mean “Yes” and the dotted arrows mean “No”.

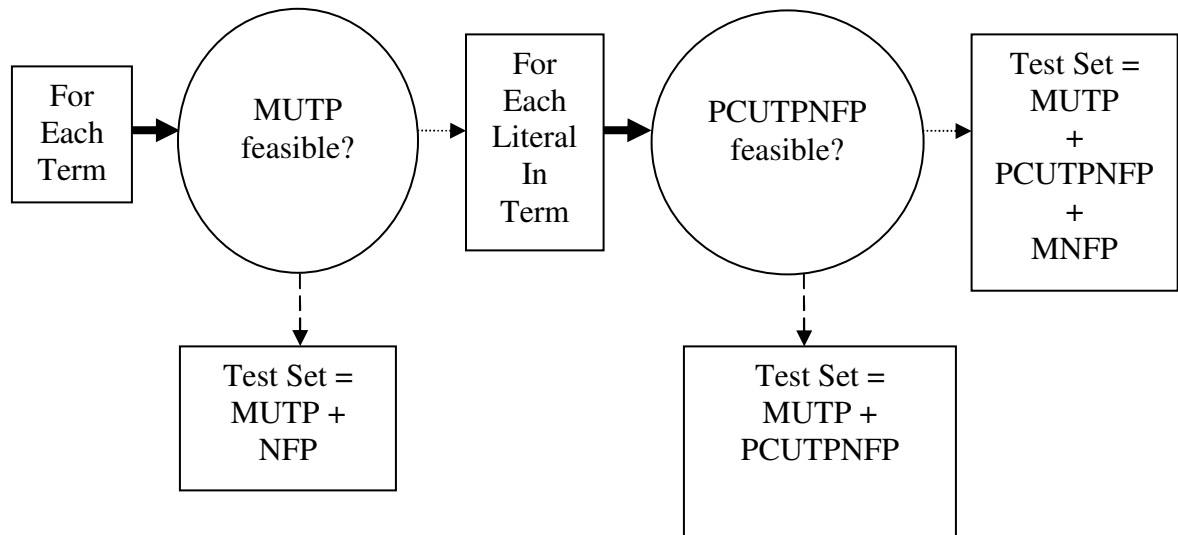


Figure 4 Minimal-MUMCUT Test Set Construction [19]

The algorithm below defines the Minimal-MUMCUT criterion for minimal DNF predicates as specified by Kaminski and Ammann [19]:

Minimal-MUMCUT Test Generation Algorithm

```

for each term X
  generate MUTP tests for X
  if the MUTP criterion is infeasible* for X
    for each literal x in X
      generate PCUTPNFP tests for x
      if the PCUTPNFP** criterion is infeasible for x, generate MNFP tests for x
    end for
  else generate an NFP for each literal x in X to overlap NFPs***
end for

```

* The MUTP criterion is infeasible for a term X if and only if an equivalent LIF exists by inserting some literal y into term X . Thus, to determine MUTP criterion feasibility, an exclusive-OR algorithm is used to evaluate all possible LIFs for equivalency. If the result of the exclusive-OR between the original predicate and the predicate with the LIF is FALSE, then the LIF is equivalent and the MUTP criterion is infeasible. However, for the algorithm above, the MUTP criterion is considered to be feasible for term X even when an equivalent LIF occurs by inserting literal y into term X as long as term X is a single-literal term or literal y occurs in a single-literal term. For example, in $ab + c$, the MUTP criterion is infeasible for term ab as literal c must be FALSE in a UTP for term ab . Thus, $ab \sim c + c$ represents an equivalent LIF. However, the MUTP criterion is still considered feasible for term ab for the algorithm since literal c is in a single-literal term.

** The PCUTPNFP criterion is infeasible for literal x if and only if an equivalent LRF results by replacing x in term X with some literal y . Thus, to determine PCUTPNFP feasibility an exclusive-OR algorithm is used to evaluate all LRFs for equivalency. If the result of the exclusive-OR between the original predicate and the predicate with the LRF is FALSE, then the LRF is equivalent and the PCUTPNFP criterion is infeasible.

*** Overlapping NFPs is a set covering combinatorial optimization problem known to be NP-complete. An heuristic is used in the algorithm above to approximate minimizing the number of NFPs generated. An example of an optimization model is in Appendix A.

As an example of satisfying the Minimal-MUMCUT criterion, consider the predicate $ab + cd$. 1101 and 1110 are UTPs for ab and the MUTP criterion is feasible for ab . 0101 and 1010 are NFPs for a and b , respectively. 0111 and 1011 are UTPs for cd and the

MUTP criterion is feasible for cd . 0101 and 1010 are NFPs for c and d , respectively. A test set is {1101, 1110, 0101, 1010, 0111, 1011}. The Minimal-MUMCUT criterion reduces test set size by overlapping NFPs when possible and only producing PCUTPNFP and MNFP tests when necessary on a literal-by-literal basis. Figure 5 updates the subsumption hierarchy shown in Figure 3 in section 2.2 to include the Minimal-MUMCUT criterion and Table 8 summarizes key aspects of the Minimal-MUMCUT criterion.

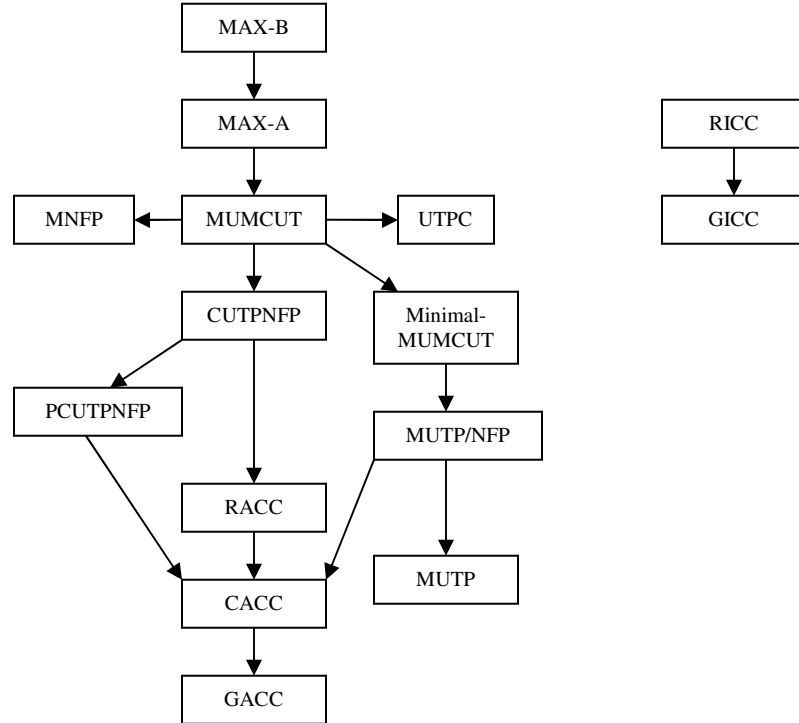


Figure 5 Updated Subsumption Hierarchy with Minimal-MUMCUT [19]

Table 8 Minimal-MUMCUT Logic Coverage Criterion Summary [19]

Test Name	Test Type	Guaranteed Faults Detected	Subsumes	Subsumed by	Minimum Test Size	Maximum Test Size
Minimal-MUMCUT	Syntactic	ENF, TNF, LNF, TOF, ORF., ORF+, LOF, LIF, LRF	MUTP/NFP, MUTP, CACC, GACC	MUMCUT, MAX-A, MAX-B	$m + 1$ to $2m + 1$ where m is the number of terms	Uncertain, but less than $2m(n-1) + \frac{mn^2}{2}$ where m is the number of terms and n is the number of literals

4.2 Test Set Size

This section describes minimum and maximum Minimal-MUMCUT test set size in theory and in practice. It is also shown in this section how test set size is related to criterion feasibility and LRF detection.

The Minimal-MUMCUT criterion always requires selecting test cases to satisfy the MUTP criterion. Minimal-MUMCUT also requires tests to satisfy either single NFP coverage for any given literal (least expensive), PCUTPNFP for any given literal, or MNFP for any given literal (most expensive), depending on criterion feasibility. The tests for MUTP and the tests involving an NFP will have no tests that overlap since for MUTP, all tests evaluate to TRUE and for any test involving an NFP, the test evaluates to FALSE. This allowed test set size for Minimal-MUMCUT to be established [24] as described next.

The maximum number of tests for Minimal-MUMCUT is simply the maximum number of tests for MUTP + the maximum number of test cases for MNFP. The maximum number of tests for MUTP is $2m(n-1)$ and the maximum number of tests for

MNFP is $\frac{1}{2}mn^2$, where m is the number of terms and n is the total number of literals (not the number of unique literals) [24]. Thus, maximum test set size of Minimal-MUMCUT is $2m(n-1) + \frac{1}{2}mn^2$.

The minimum number of tests for Minimal-MUMCUT is simply the minimum number of tests for MUTP + the minimum number of tests for NFP coverage, or $m + l$. The minimum test size of $m + l$ will arise when the predicate has one literal for each term, such as $a + b + c$. In this case, there are three terms so the minimum number of tests is four. Three of these tests come from MUTP: (100, 010, 001) and one comes from a single NFP: 000.

Chen, Lau and Yu [8] evaluated MUMCUT test set size (using the greedy MUTP algorithm developed by Chen and Lau [6]) for 19 minimal DNF predicates from an air traffic collision avoidance system (TCAS). There were originally 20 predicates but number 12 was excluded due to a missing a right parenthesis detected by Weyuker et al. [46]. The predicates have from 5 to 13 unique literals (see Appendix B). Kaminski and Ammann [19] created Minimal-MUMCUT tests for each predicate and assessed MUTP feasibility for each term and PCUTPNFP feasibility for each literal. The Minimal-MUMCUT algorithm presented earlier was implemented in Java to obtain the results.

The results showed that the PCUTPNFP criterion was feasible for every literal (853 of them), so the expensive MNFP tests were not needed for any literal. For 204 literals (23.92%), the MUTP criterion was feasible for the literal's term and thus MUTP tests detect an LRF, meaning the less expensive approach of requiring NFP tests instead of PCUTPNFP tests is used. For the other 649 literals (76.08%), PCUTPNFP tests were

needed to detect an LRF. For four predicates, the MUTP criterion was feasible for every term, so PCUTPNFP tests were not needed for any literal for these four predicates. For 16 predicates, the MUTP criterion was feasible for at least one term. Thus, PCUTPNFP tests were not needed for literals in at least one term in most predicates. On average, Minimal-MUMCUT test set size was 2.40% of exhaustive test set size. Table 9 displays feasibility and LRF detection results and Table 10 displays test set size results.

Table 9 Criterion Feasibility and LRF detection [19]

Predicate	Number of terms that are MUTP feasible	Number of terms that are MUTP infeasible	Number of literals for which MUTP detects LRF	Number of literals needing PCUTPNFP to detect LRF
1	1	4	5	24
2	4	9	33	72
3	2	23	10	136
4	1	2	1	6
5	1	8	1	27
6	2	4	22	36
7	4	4	28	32
8	4	0	32	0
9	2	0	14	0
10	0	6	0	60
11	1	8	6	57
*12	N/A	N/A	N/A	N/A
13	0	6	0	14
14	0	6	0	16
15	1	10	2	30
16	1	22	2	85
17	2	4	8	24
18	2	6	8	30
19	4	0	20	0
20	2	0	12	0
Sum	34	122	204	649

* number 12 excluded due to a missing a right parenthesis

Table 10 Minimal-MUMCUT Test Set Size [19]

Predicate	Minimal-MUMCUT [19]	2^n	Percentage
1	27	128	21.09%
2	81	512	15.82%
3	148	4096	3.61%
4	9	32	28.13%
5	34	512	6.64%
6	62	2048	3.03%
7	62	1024	6.05%
8	36	256	14.06%
9	16	128	12.50%
10	62	8192	0.76%
11	61	8192	0.74%
12	N/A	N/A	N/A
13	17	4096	0.42%
14	22	128	17.19%
15	39	512	7.62%
16	104	4096	2.54%
17	39	2048	1.90%
18	48	1024	4.69%
19	16	256	6.25%
20	14	128	10.94%
Sum	897	37,408	
Avg	47.21	1968.84	2.40%

4.3 Single Minimal DNF Fault Detection

This section focuses on the single minimal DNF fault detection capability of the Minimal-MUMCUT criterion. It highlights how criterion feasibility is linked to equivalent faults and fault detection, with a special focus on the LIF and the LRF.

Any logic coverage criterion that includes at least one UTP for each term and one NFP for each literal is guaranteed to detect all fault types in Lau and Yu's fault hierarchy

except for the LIF and the LRF [30]. The Minimal-MUMCUT criterion meets this requirement. To understand why a test set that satisfies the Minimal-MUMCUT criterion also guarantees LIF and LRF detection for a minimal DNF predicate, it is important to first understand criterion feasibility and equivalent faults.

The LIF can result in an equivalent fault in that no input can distinguish the original predicate from the faulty version. For example, consider $ab + bc$ and the LIF where $\sim c$ is inserted into the first term to yield $ab\sim c + bc$. This is an equivalent LIF because the original predicate and faulty predicate evaluate to the same value for all inputs. To make term ab true and term $ab\sim c$ false, the point 111 can be used, but doing so makes term bc true in each predicate. AN LIF will be equivalent if and only if the MUTP criterion is infeasible. The MUTP criterion is infeasible for term ab as literal c must be 0 in a UTP for term ab . The LRF can also result in an equivalent fault. For example, consider $ab + b\sim c + \sim bc$ and the LRF where literal c replaces literal b in the first term to yield $ac + b\sim c + \sim bc$. To make term ab true and term ac false, the point 110 can be used, but doing so makes term $b\sim c$ true in each predicate. To make term ab false and term ac true, the point 101 can be used, but doing so makes term $\sim bc$ true in each predicate. AN LRF will be equivalent if and only if the PCUTPNFP criterion is infeasible (which also means that the CUTPNFP criterion is infeasible). The PCUTPNFP criterion is infeasible for literal b in term ab as the only UTP for term ab is 111 and the only NFP for literal b in term ab is 100 which differs from the UTP of 111 in both the value of b and c (c must be 1 in a UTP for term ab).

The condition for detecting an LIF is as follows as specified by Lau and Yu [30]. If some literal not intended to be in term X is inserted into X as itself or as its negation, then a set of UTPs for X , where all literals not in X attain the values 0 and 1, detects the fault. MUTP tests are guaranteed to detect an LIF. However, when the MUTP criterion is infeasible, an LRF exists that MUTP tests may not detect. Consider $ab + ac + bc$ and an LIF producing $ab\sim c + ac + bc$. The MUTP criterion is infeasible for ab as the only UTP for ab is 110. Therefore, MUTP tests do not detect the corresponding LRFs: $\sim cb + ac + bc$ and $a\sim c + ac + bc$.

MUTP tests are guaranteed to detect an LRF for a literal if the MUTP criterion is feasible for that literal's term. In this case, it is only necessary to satisfy the MUTP criterion and the NFP criterion (an NFP for each literal in the term) to guarantee detecting all fault types in Lau and Yu's fault hierarchy for that term [24]. Thus, neither the PCUTPNFP nor the MNFP criterion is needed for literals in a MUTP feasible term to detect LRFs for literals in that term. The NFP for a literal in a MUTP feasible term can overlap with NFPs for other literals in other terms since any NFP for a literal detects an LOF for that literal. If a term is MUTP infeasible but all literals in the predicate external to that term that cannot be 0 or 1 in a UTP for the term exist in single-literal terms, LRF detection is still guaranteed by MUTP tests. The reason is that an LRF involving replacing a literal with a literal (or its negation) that exists in a single-literal term results in a TOF, LOF, or a TRUE predicate. Since a UTP guarantees detection of a TOF and an NFP guarantees detecting an LOF or a fault where the predicate is stuck at 1, MUTP tests supplemented with NFPs guarantee LRF detection. For example, in $a + b$, replacing a

with b results in a TOF for a and replacing a with $\sim b$ makes the predicate TRUE. In $ab + c$, replacing a with c results in a TOF for ab and replacing a with $\sim c$ results in an LOF for a .

The condition for detecting an LRF is as follows as specified by Lau and Yu [30] and Kaminski and Ammann [22]. If literal x in X is wrongly implemented as some other literal or the negation of some other literal not in X , then any of the following detects the fault: a set of UTPs for X where all literals not in X attain the values 0 and 1; a set of NFPs for x where all literals not in X attain the values 0 and 1; a UTP-NFP pair where the points differ only in the value of x and possibly in the values of all literals that can be 0 or 1 in a UTP for term X . The PCUTPNFP criterion is designed to produce tests that detect an LRF but fails to do so when it is infeasible. However, when the PCUTPNFP criterion is infeasible, MNFP tests can be added to guarantee LRF detection. Thus, only when both the MUTP criterion is infeasible and the PCUTPNFP criterion is infeasible are MNFP tests needed to guarantee LRF detection. Consider $abc + abd + \sim b\sim d + \sim de$. The PCUTPNFP criterion is infeasible for b in abc . The only UTP for abc is 11100 so the only way to try to satisfy the PCUTPNFP is to satisfy the CUTPNFP criterion. A corresponding NFP of 10100 is not possible for b in abc because this is a TRUE point. Now consider the LRF $a\sim ec + abd + \sim b\sim d + \sim de$. Since the PCUTPNFP criterion is infeasible for b in abc , this LRF goes undetected by PCUTPNFP tests. A single NFP for b in abc is not guaranteed to detect the LRF either. The point 10111 is an NFP for b in abc , but this point fails to detect the LRF. The MNFP criterion requires that the NFP 10110 be chosen for b in abc , detecting the LRF.

Figure 6 displays Lau and Yu's Fault Hierarchy modified based on how criterion feasibility affects fault detection as indicated by Kaminski and Ammann [19]. A solid arrow from a source fault to a destination fault indicates that if a test detects a source fault, it also detects a corresponding destination fault. When the MUTP criterion is infeasible, a test set detecting all LIFs is not guaranteed to detect all LRFs. Thus the solid arrow between the LIF and LRF in Lau and Yu's hierarchy is changed to a dashed arrow. In Lau and Yu's hierarchy no arrow exists between the LRF and LOF. A dashed arrow is added to represent that when guaranteeing detection of all LIFs does not guarantee detection of all LRFs (due to MUTP infeasibility), adding tests to detect the undetected LRFs will detect all corresponding LOFs (unless the PCUTPNFP criterion is infeasible). The reason is that when the MUTP criterion is infeasible but the PCUTPNFP criterion is feasible, a UTP will not detect an LRF but a corresponding NFP will. Since the Minimal-MUMCUT criterion always requires MUTP tests, the LIF is guaranteed to be detected. Since the Minimal-MUMCUT requires (1) PCUTPNFP tests when the MUTP criterion is infeasible and the PCUTPNFP criterion is feasible and (2) MNFP tests when the PCUTPNFP criterion is infeasible, LRF detection is guaranteed.

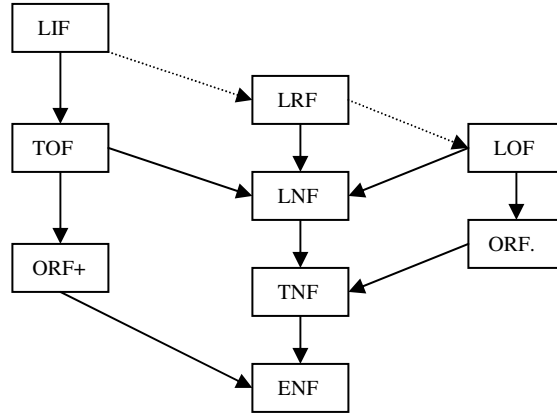


Figure 6 Fault Hierarchy Based on Infeasibility [19]

4.4 Double Minimal DNF Fault Detection

A double minimal DNF fault occurs in a predicate when two faults represented in the fault types in Figure 6 are introduced. This section focuses on the double minimal DNF fault detection capability of the Minimal-MUMCUT criterion. It highlights how criterion feasibility is linked to double fault detection, with a special focus on a double fault involving two LIFs.

Any two single faults in Lau and Yu's hierarchy can be combined to form a double fault. Lau, Liu and Yu [27, 28, 29] show that 92 double fault types exist when considering order and the different semantic versions that can occur. The reason why 92 double fault types exist (as opposed to 81) is that different semantic versions occur depending on whether or not both faults occur in the same or different terms. However, order only causes a semantic difference in double fault types for eight cases. This is because some ordered double fault types are equivalent to each other. Some examples are given next.

Consider the expression $ab + cd + ef$ and the LNF where literal a is negated and the TNF where the second term is negated. Whether or not the LNF or TNF occurs first does not matter as both result in $\sim ab + \sim(cd) + ef$. Now consider the LIF where literal e is inserted into term ab and the TNF where the first term is negated. If the LIF occurs first, the result is $\sim(abe) + cd + ef$. If the TNF occurs first, the result is $\sim(ab)e + cd + ef$, which is semantically different. Furthermore, certain double fault types result in different versions depending on whether the faults occur in the same or different terms. For example, a double fault where two LRFs occur in the same term can be considered different than a double fault where two LRFs occur in different terms.

The result is that there are 92 double fault types when considering order and versions, 82 double fault types when considering versions but not ordering, 53 double fault types when considering order but not versions and 45 double fault types when considering neither order nor versions [27, 28, 29]. In addition to negating the entire predicate, Lau et al. [27, 28, 29] consider the ENF to include negating the disjunction of two or more terms (meaning changing $ab + cd + ef$ to $ab + \sim(cd + ef)$). This definition prevents the ENF-ENF double fault from resulting in a faulty predicate that is equivalent semantically to the original predicate.

Lau et al. [27, 28, 29] state that BASIC tests (selecting a UTP for each term and an NFP for each literal) detect all but 8 of the 92 types when considering order and versions and all but 6 of the 45 types when considering neither order nor versions. The Minimal-MUMCUT criterion subsumes the BASIC criterion, but it does not subsume the additional criteria proposed by Lau et al. [27, 28, 29] needed to guarantee detection of all

double fault types. Below are the eight double fault types that Minimal-MUMCUT tests are not guaranteed to detect as specified by Kaminski and Ammann [20].

1. TOF-LRF:

Intended: $\sim a \sim b + ab + cd + \sim c \sim d$

Actual: $ab + \mathbf{ad} + \sim c \sim d$

2. ORF.-LRF where faults occur in different terms:

Intended: $\sim a \sim b + ab + cd + \sim c \sim d$

Actual: $\sim a \sim b + \mathbf{abcd} + \mathbf{a} \sim d$

3. LOF-LRF where faults occur in same term:

Intended: $abc + abd$

Actual: $\mathbf{dc} + abd$

4. LIF-LIF where faults occur in different terms:

Intended: $ab + bc$

Actual: $ab \sim \mathbf{c} + bc \sim \mathbf{a}$

5a. LIF-LRF where faults occur in different terms:

Intended: $abc + cde$

Actual: $abcd + \mathbf{cbe}$

5b. LIF-LRF where faults occur in same term:

Intended: $abc + cde$

Actual: $ab\mathbf{de} + cde$

6a. LRF-LRF where faults occur in different terms:

Intended: $abcd + abef$

Actual: $\mathbf{ebcd} + \mathbf{acef}$

6b. LRF-LRF where faults occur in same term:

Intended: $abcd + abef$

Actual: $\mathbf{efcd} + abef$

Kaminski and Ammann [20] prove that if the MUTP criterion is feasible for a term, Minimal-MUMCUT tests guarantee detecting all double faults involving that term or literals in that term. This proof was done by taking the conditions needed for double

fault detection as specified by Lau et al. [27, 28, 29], translating them in terms of criterion feasibility and then showing that if the MUTP criterion is feasible, the eight fault types mentioned above can all be detected by a MUTP test set.

Kaminski and Ammann [20] also prove that if the MUTP criterion is infeasible for a term, but the CUTPNFP criterion is feasible for a literal in the term (and hence the PCUTPNFP criterion is also feasible), Minimal-MUMCUT tests guarantee detection of all double faults for the literal and term, except the LIF-LIF. This proof was done by observing that when the MUTP criterion is infeasible for a term, the Minimal-MUMCUT criterion subsumes the PCUTPNFP criterion for each literal in that term. The proof establishes that that when the PCUTPNFP criterion is feasible, PCUTPNFP tests detect the eight double fault types mentioned above except the LIF-LIF.

Kaminski and Ammann [20] also prove that if the PCUTPNFP criterion is infeasible for a literal, Minimal-MUMCUT tests guarantee detecting all but eight double fault types. This proof is accomplished by first proving valid and invalid feasibility combinations amongst the MUTP, PCUTPNFP and MNFP criteria. These combinations are displayed in Table 11.

Table 11 Criterion Feasibility Combinations [20]

Row	MUTP feasible	PCUTPNFP feasible	MNFP feasible	Valid
1	No	No	No	Yes
2	No	No	Yes	No
3	No	Yes	No	Yes
4	No	Yes	Yes	No
5	Yes	No	No	No
6	Yes	No	Yes	No

Row	MUTP feasible	PCUTPNFP feasible	MNFP feasible	Valid
7	Yes	Yes	No	Yes
8	Yes	Yes	Yes	Yes

From Table 11, when the PCUTPNFP criterion is infeasible for a literal, the MUTP criterion is infeasible for that literal's term, so MUTP tests do not guarantee detecting any of the eight double fault types BASIC tests do not guarantee detecting. Also from Table 11, when the PCUTPNFP criterion is infeasible for a literal, the MNFP criterion is infeasible for that literal, so MNFP tests do not guarantee detecting any of the eight double fault types BASIC tests do not guarantee detecting. So if the PCUTPNFP criterion is infeasible, the eight double fault types that BASIC tests do not guarantee detecting are not guaranteed to be detected by Minimal-MUMCUT tests. Table 12 summarizes these results.

Table 12 Double Fault Detection of Minimal-MUMCUT Tests Based on Criterion Feasibility [20]

Row	MUTP feasible	PCUTPNFP feasible	MNFP feasible	Double Fault Types Missed
1	No	No	No	8
2	No	No	Yes	N/A
3	No	Yes	No	1
4	No	Yes	Yes	N/A
5	Yes	No	No	N/A
6	Yes	No	Yes	N/A
7	Yes	Yes	No	0
8	Yes	Yes	Yes	0

Table 12 indicates that for a Minimal-MUMCUT test set:

- 1) If the MUTP criterion is feasible for a term, all double faults involving the term or its literals are detected.
- 2) If the MUTP criterion is infeasible for a term, but the PCUTPNFP criterion is feasible for a literal in that term, all but one double fault type (the LIF–LIF) involving that literal or that literal’s term is detected.
- 3) If the PCUTPNFP criterion is infeasible for a literal, all but eight double fault types involving that literal or that literal’s term are detected.

The above results indicate that the LIF-LIF is the most common double fault to be undetected by the Minimal-MUMCUT criterion because it will go undetected whenever the MUTP criterion is infeasible. To evaluate how often Minimal-MUMCUT tests are likely to miss detecting the eight double fault types in practice, an empirical evaluation was performed using the same sample of 19 predicates described in chapter 4 as well as an additional sample of 275 minimal DNF predicates (each containing at least 5 unique literals) in avionics software. Although the MUTP criterion was not feasible for every term in these predicates, the PCUTPNFP criterion was feasible for every literal in every predicate (for both the 19 original predicates and for the additional 275 predicates). Thus, Minimal-MUMCUT tests detected all double fault types except the LIF-LIF for all of these predicates. For the additional 275 predicates, the MUTP criterion was feasible for 98% of them, meaning that Minimal-MUMCUT tests guaranteed detecting all double fault types for 98% of the 275 predicates.

Lau et al. [27, 28, 29] developed the Supplementary Multiple Overlapping True Point (SMOTP) criterion to detect the LIF-LIF. This criterion requires that for each pair of terms, a set of OTPs be included such that all literals not in either term are assigned the values 0 and 1. However, the LIF-LIF can only go undetected by Minimal-MUMCUT tests when both terms involved in the double fault are MUTP infeasible. Thus, the SMOTP criterion only needs to be included for a subset of the possible pairs of terms in the predicate. So incorporating one additional criterion into the Minimal-MUMCUT criterion guarantees detecting all double faults in the predicates examined and this criterion is not needed for all pairs of terms. Lau et al. [27, 28, 29] developed five other criteria that guarantee complete double fault detection, but satisfying these criteria is expensive and none were necessary for double fault detection in the predicates examined. Kaminski and Ammann [20] showed how the Minimal-MUMCUT criterion can be modified to include the SMOTP criterion as follows:

Minimal-MUMCUT and SMOTP Test Generation Algorithm

```

for each term X
  generate MUTP tests for X
  if the MUTP criterion is infeasible for X
    for each MUTP infeasible term Y
      generate SMOTP tests for X and Y
    end for
  for each literal x in X
    generate PCUTPNFP tests for x
    if the PCUTPNFP criterion is infeasible for x, generate MNFP tests for x
  end for
  else generate an NFP for each literal x in X to overlap NFPs
end for

```

On average, 6.79 non-equivalent LIF-LIFs per predicate went undetected by Minimal-MUMCUT tests amongst the 19 predicates in the study. For four predicates (8,

9, 19, 20) the MUTP criterion was feasible for every term so Minimal-MUMCUT tests detected all double faults. For 16 predicates, the MUTP criterion was feasible for at least one term so Minimal-MUMCUT tests detected all double faults in at least one term (and its literals) in these predicates. For two predicates (7 and 17), every pairing of two equivalent LIFs resulted in an equivalent LIF-LIF. Thus, Minimal-MUMCUT tests detected all non-equivalent double faults for these predicates.

An example of an LIF-LIF that Minimal-MUMCUT tests did not detect is mutating $a\sim bd + a\sim cd + e$ to $a\sim bdc + a\sim cdb + e$. Detection requires one additional test beyond what the Minimal-MUMCUT criterion requires. 10010 makes the first two terms in the original predicate TRUE and the first two terms in the faulty predicate FALSE. Table 13 shows the number of undetected LIF-LIFs (and thus the maximum number of extra tests needed to guarantee detection), as well as the percentage undetected. 99.91% of LIF-LIFs were detected so few additional tests are needed to detect all non-equivalent LIF-LIFs.

Table 13 LIF-LIFs Undetected by Minimal-MUMCUT Tests [20]

Predicate	Number of undetected LIF-LIFs	Total Number LIF-LIFs	Percentage undetected
1	2	66	3.03%
2	3	276	1.09%
3	34	42,278	0.08%
4	1	120	0.83%
5	10	5,565	0.18%
6	2	120	1.67%
7	0	780	0.00%
8	0	0	N/A
9	0	0	N/A

Predicate	Number of undetected LIF-LIFs	Total Number LIF-LIFs	Percentage undetected
10	6	630	0.95%
11	3	5,778	0.05%
12	N/A	N/A	N/A
13	4	6,670	0.06%
14	5	1,326	0.38%
15	17	8,911	0.19%
16	38	71,253	0.05%
17	0	2,278	0.00%
18	4	3,486	0.11%
19	0	276	0.00%
20	0	6	0.00%
Sum	129	149,819	
Average	6.79	7,885.21	0.09%

The number of LIFs for a predicate with m terms, n unique literals and n_i literals in term i is $L = 2 \sum_{i=1}^m (n - n_i)$ (multiplication by 2 since each unique literal in the predicate that is not in the term of interest may be inserted as itself or as its negation). The number of possible LIF-LIFs is L^2 . However, this number is smaller in Table 13 because the order of each LIF in an LIF-LIF is irrelevant and a single LIF paired with itself defaults to a single LIF. Thus, in Table 13 the total number of LIF-LIFs is $L * (L-1) / 2$.

When two LIFs combine to form an LIF-LIF, four combinations exist for the equivalency relationship between the faulty and non-faulty predicates as Table 14 shows.

Table 14 Equivalency Relationships Between Faulty and Non-Faulty Predicates [20]

LIF 1	LIF 2	LIF-LIF
Not equivalent	Not equivalent	Not equivalent
Not equivalent	Equivalent	Not equivalent
Equivalent	Not Equivalent	Not equivalent
Equivalent	Equivalent	Undetermined

MUTP tests are guaranteed to detect an LIF. If the MUTP criterion is feasible for either term in the LIF-LIF, the LIF-LIF is detected by MUTP tests [20]. In the first three rows of Table 14, the first or second LIF is not equivalent, so the MUTP criterion is feasible for at least one term. Thus, MUTP tests also detect the LIF-LIF, meaning the LIF-LIF is non-equivalent. To show how two equivalent LIFs form a non-equivalent LIF-LIF, consider $a \sim bd + a \sim cd$. Inserting c into $a \sim bd$ results in an equivalent LIF, as does inserting b into $a \sim cd$. Combining the equivalent LIFs results in a non-equivalent LIF-LIF: $a \sim bdc + a \sim cdb$ (detected by 1001 – the original evaluates to TRUE but the faulty version evaluates to FALSE). To show how two equivalent LIFs form an equivalent LIF-LIF, consider $a \sim bd + a \sim cd + e$. Inserting $\sim e$ into $a \sim bd$ results in an equivalent LIF, as does inserting $\sim e$ into $a \sim cd$. The equivalent LIF-LIF is $a \sim bd \sim e + a \sim cd \sim e + e$.

Table 14 shows an LIF-LIF can only be equivalent when each LIF is equivalent. However, when each LIF is equivalent, an LIF-LIF can also be non-equivalent. When the MUTP criterion is feasible for either term where an LIF occurs, Minimal-MUMCUT tests will detect an LIF-LIF [20]. However, when the MUTP criterion is infeasible for both terms in an LIF-LIF (meaning each LIF is equivalent), a non-equivalent LIF-LIF will go undetected by Minimal-MUMCUT tests [20]. So if most LIF-LIFs formed from

two equivalent LIFs are equivalent, there will be few non-equivalent LIF-LIFs that a Minimal-MUMCUT test fails to detect (fault coupling will be rare). On the other hand, if most such LIF-LIFs are non-equivalent, there will be more non-equivalent LIF-LIFs that a Minimal-MUMCUT test fails to detect (fault coupling will be common). Table 15 shows the percentage of LIF-LIFs that were equivalent based on combining two equivalent LIFs. In Table 15 the column “Number of equivalent LIF-LIFs” refers to the number of equivalent double faults where each of the single faults is an equivalent LIF. The column “Number of equivalent LIF – equivalent LIF pairs” refers to the number of double faults where each of the single faults is an equivalent LIF.

Table 15 Equivalent LIF-LIFs as a Percentage of Equivalent LIF Pairings [20]

Predicate	Number of equivalent LIF-LIFs	Number of equivalent LIF-equivalent LIF pairs	Percentage equivalent LIF-LIFs
1	4	6	66.67%
2	78	81	96.30%
3	2177	2211	98.24%
4	5	6	83.33%
5	518	528	98.11%
6	4	6	66.67%
7	28	28	100.00%
8	0	0	N/A
9	0	0	N/A
10	60	66	90.91%
11	63	66	95.45%
12	N/A	N/A	N/A
13	132	136	97.06%
14	61	66	92.42%
15	803	820	97.93%
16	4427	4465	99.15%
17	12	12	100.00%

Predicate	Number of equivalent LIF-LIFs	Number of equivalent LIF-equivalent LIF pairs	Percentage equivalent LIF-LIFs
18	87	91	95.60%
19	0	0	N/A
20	0	0	N/A
Sum	8459	8588	
Average	445.21	452.00	98.50%

In Table 15, the number of equivalent LIF – equivalent LIF pairs is $L * (L-1) / 2$ given L equivalent LIFs as order and pairing an equivalent LIF with itself are not considered. The results show a large percentage of LIF-LIFs formed from two equivalent LIFs were equivalent. Thus, few non-equivalent LIF-LIFs go undetected by Minimal-MUMCUT tests and fault coupling is rare. Polo, Piattini and Garcia-Rodriguez [39] state that two equivalent single faults always result in an equivalent double fault. This is incorrect for LIFs. However, Table 15 shows that it is likely that two equivalent LIFs form an equivalent LIF-LIF.

Table 16 compares test set size (based on the 19 original predicates) for the Minimal-MUMCUT criterion with a test set supplemented with tests needed to detect LIF-LIFs. The number of tests needed to detect LIF-LIFs is less than the number of undetected LIF-LIFs because multiple undetected LIF-LIFs can sometimes be detected by the same test. 129 LIF-LIFs went undetected (an average of 6.79 per predicate) but 108 additional tests can be used to detect them (an average of 5.68 tests per predicate). On average, Minimal-MUMCUT test set size is 89.25% of the test set size formed by

combining Minimal-MUMCUT tests and tests to detect all LIF-LIFs. By adding on average an additional 5.68 tests to a Minimal-MUMCUT test set, all LIF-LIFs can be detected in these predicates.

Table 16 Minimal-MUMCUT vs. Minimal-MUMCUT + LIF-LIF Test Set Size [20]

Predicate	Minimal – MUMCUT [19]	Minimal – MUMCUT + LIF-LIF Tests [20]	Percentage	2^n
1	27	29	93.10%	128
2	81	84	96.43%	512
3	148	173	85.55%	4096
4	9	10	90.00%	32
5	34	41	82.93%	512
6	62	64	96.88%	2048
7	62	62	100.00%	1024
8	36	36	100.00%	256
9	16	16	100.00%	128
10	62	68	91.18%	8192
11	61	64	95.31%	8192
12	N/A	N/A	N/A	N/A
13	17	21	80.95%	4096
14	22	27	81.48%	128
15	39	56	69.64%	512
16	104	133	78.20%	4096
17	39	39	100.00%	2048
18	48	52	92.31%	1024
19	16	16	100.00%	256
20	14	14	100.00%	128
Sum	897	1005		37,408
Avg	47.21	52.89	89.25%	1968.84

4.5 General Logic Fault Detection

This section begins with an examination of the frequency of minimal DNF predicates in software. This is important because for syntactic criteria, fault detection that holds for minimal DNF predicates does not in general hold for non-minimal DNF predicates. Next this section explores changes to the Minimal-MUMCUT criterion to address minimal CNF. The section concludes with a discussion of changes to the Minimal-MUMCUT criterion to handle the case when neither minimal DNF nor minimal CNF holds.

For syntactic criteria assuming minimal DNF, it is important to know what types of software have predominantly minimal DNF predicates. Chilenski [9] found that 95% of 20,256 predicates in avionics software were in minimal DNF. However, when a predicate contains less than three unique literals the author conjectures that exhaustive testing is best because it is only at three unique literals that the Minimal-MUMCUT criterion begins to potentially offer a 50% test set size savings over exhaustive coverage. This raises the questions of what types of software generally have predicates with at least three unique literals and what proportion of such predicates are in minimal DNF. Chilenski and Miller [10] report that avionics software often has predicates with many unique literals and Chilenski [9] extracted a predicate with 77 unique literals. Thus, the Minimal-MUMCUT criterion should be useful for testing avionics software. In terms of predicate format for large predicates, Kaminski and Ammann [22] report that 3% of the 20,256 predicates Chilenski examined contain five or more unique literals, but 80% of these predicates are in minimal DNF. Thus, fault detection is guaranteed for the majority of

these predicates by Minimal-MUMCUT tests for single and double faults in Lau and Yu's fault hierarchy.

Minimal CNF

Kaminski and Ammann [22] report that of the 3% of the 20,256 predicates Chilenski examined that contain five or more unique literals, 85% were either in minimal DNF or minimal CNF (or both). Thus, modifying the Minimal-MUMCUT criterion to incorporate predicates in minimal CNF (an approach known as Union Minimal-MUMCUT) can guarantee fault detection of all single and 84 of 92 double faults in Lau and Yu's fault hierarchy for 85% of these predicates. An initial exploratory study by students in a graduate class at George Mason University of open source software found that of 43 predicates that contained three or more unique literals the following held:

14 were in minimal CNF but not minimal DNF

1 was in minimal DNF but non minimal CNF

40 were in minimal CNF

27 were in minimal DNF

41 were in either minimal CNF or minimal DNF

2 were in neither minimal CNF nor minimal DNF

These results show that 63% of the predicates were in minimal DNF and 95% of the predicates were in minimal DNF or minimal CNF (or both). Thus, modifying Minimal-MUMCUT to guarantee fault detection for minimal CNF predicates increases the

percentage of predicates for which fault detection is guaranteed from 63% to 95% for the single and double fault types in Lau and Yu's fault hierarchy.

Due to the dual nature of minimal CNF and minimal DNF, it is possible to modify the Minimal-MUMCUT criterion to also guarantee fault detection when a predicate is in minimal CNF but not minimal DNF. Furthermore, it is possible to modify the Minimal-MUMCUT criterion to improve its fault detection when a predicate is in neither minimal CNF nor minimal DNF as explained next. The next paragraphs introduce a new set of logic coverage criteria and a new fault hierarchy based on minimal CNF. The following new criteria assume a predicate is in minimal CNF. An example of $(a + b)(c + d)$ is used in each.

Multiple Unique False Point (MUFP) [22]: Given a minimal CNF predicate, form tests for a UFP for each term such that all literals not in the term attain values 1 and 0. A UFP for the first term must have $a=0, b=0$. Needed tests for c and d to each = 0 and 1 are 0001 and 0010. A UFP for the second term must have $c=0$ and $d=0$. Needed tests for a and b to each = 0 and 1 are 0100 and 1000. A test set is {0001, 0010, 0100, 1000}.

Multiple Near True Point [22]: Given a minimal CNF predicate, form tests for an NTP of each literal such that all literals not in the literal's term attain values 1 and 0. NTPs for a and b so that c and d each equal 0 and 1 are 1001, 1010, 0101, and 0110. Needed NTPs for c and d so that a and b each equal 0 and 1 are 0110, 1010, 0101, and 1001. A test set is {1001, 1010, 0101, 0110}.

Corresponding Unique False Point Near True Point (CUFPNTP) [22]: Given a minimal CNF predicate, for each literal in each term find a UFP and NTP such that only

the literal changes value (all other literals must be fixed). A UFP for the first term must have $a=0$, $b=0$. If $c=0$ and $d=1$, tests for literals in ab are 0001, 1001, and 0101. A UFP for the second term must have $c=0$, $d=0$. If $a=1$ and $b=0$, tests for literals in cd are 1000, 1010, and 1001. A test set is {0001, 1001, 0101, 1000, 1010}.

Partial-Corresponding Unique False Point Near True Point (PCUFPNTP) [22]:

Given a minimal CNF predicate, for each literal in each term find a UFP and NTP such that the literal changes value and the only literals that must be fixed are literals that must be fixed in a UFP for the term of interest. This criterion is more flexible than CUFPNTP and is subsumed by it (any CUFPNTP test set is also a PCUFPNTP test set). For term $(a + b)$ a MUFP test set is {0001, 0010}. To satisfy CUFPNTP for literal a , a corresponding NTP of 1001 or 1010 must be chosen. However, PCUFPNTP permits 1011 to be chosen as the NTP. 1011 differs from each UFP in either a and c or a and d . Thus, 1011 is not a corresponding NTP but it can still be chosen to satisfy PCUFPNTP because literals c and d can be 0 or 1 in a UFP for term ab . While PCUFPNTP does not offer any test set size savings over CUFPNTP for the example of $(a + b)(c + d)$, it can for other predicates because it allows greater flexibility in choosing NTPs so that they can overlap.

Figure 7 places the minimal CNF logic coverage criteria in an updated subsumption hierarchy and Table 17 gives a summary of the minimal CNF logic coverage criteria.

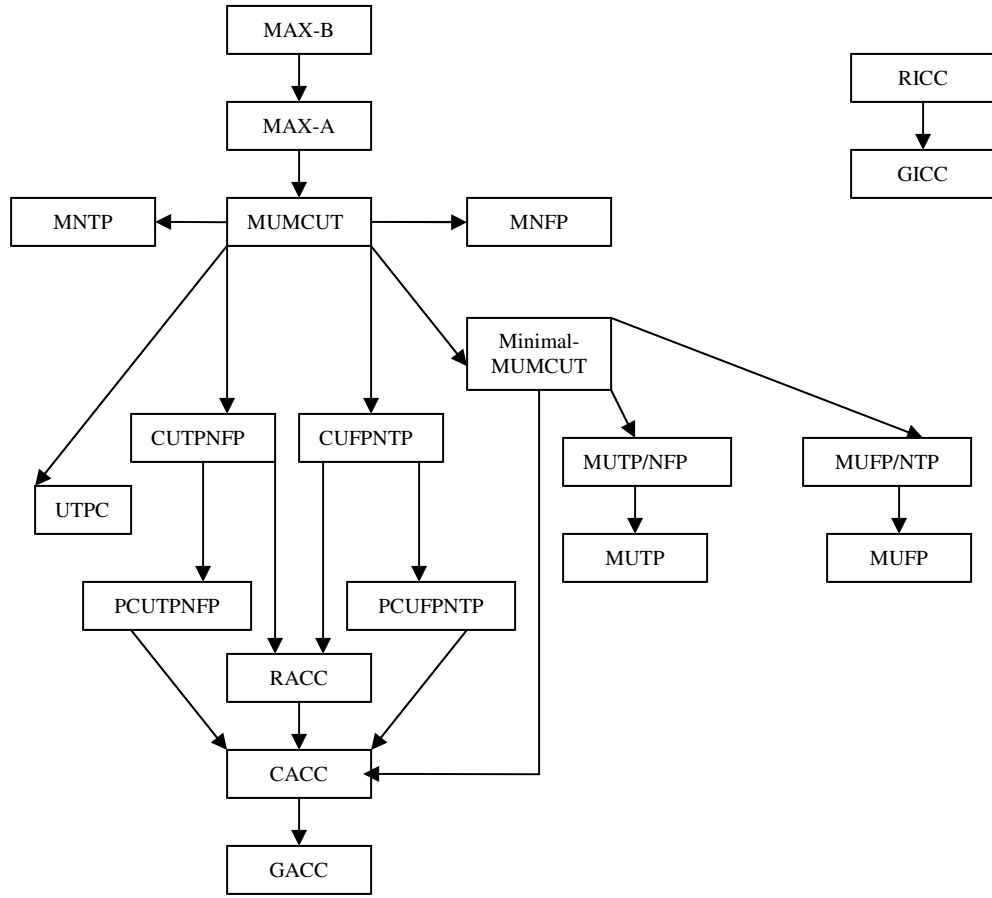


Figure 7 Updated Subsumption Hierarchy with Minimal CNF Logic Criteria [22]

Table 17 Minimal CNF Logic Criteria Summary [22]

Test Name	Test Type	Guaranteed Faults Detected	Subsumes	Subsumed by	Minimum Test Size	Maximum Test Size
Multiple Unique False Point (MUFP)*	Syntactic	ENF, TNF, LNF, TOF, ORF., LIF	-	Minimal-MUMCUT, MUMCUT, MUFP/NTP, MAX-A, MAX-B	m to $2m$ where m is the number of terms	$2m(n-1)$ where m is the number of terms and n is the number of literals
Corresponding Unique False Point Near True Point (CUFPNTP)*	Syntactic	ENF, TNF, LNF, TOF, ORF., ORF+, LOF	PCUFNTP, RACC, CACC, GACC	MUMCUT, MAX-A, MAX-B	$\sum_{i=1}^m n_i + 1$ where n_i is the number of literals in term i and m is the number of terms	$2mn$ where m is the number of terms and n is the number of literals

Test Name	Test Type	Guaranteed Faults Detected	Subsumes	Subsumed by	Minimum Test Size	Maximum Test Size
Partial Corresponding Unique False Point Near True Point (PCUFPNTP)**	Syntactic	ENF, TNF, LNF, TOF, ORF., ORF+, LOF	CACC, GACC	CUFPNTP, MUMCUT, MAX-A, MAX-B	$\sum_{i=1}^m n_i + 1$ where n_i is the number of literals in term i and m is the number of terms	$2mn$ where m is the number of terms and n is the number of literals
Multiple Near True Point (MNTP)*	Syntactic	ENF, TNF, LNF, ORF+, LOF	-	MUMCUT, MAX-A, MAX-B	When infeasibilities arise: 1. Uncertain otherwise.	$\frac{mn^2}{2}$ where m is the number of terms and n is the number of literals
Multiple Unique False Point / Near True Point (MUFP / NTP) *	Syntactic	ENF, TNF, LNF, TOF, ORF, LOF, LIF	CACC, GACC, MUFP	MUMCUT, MAX-A, MAX-B	$m + 1$ to $2m + 1$ (where m is the number of terms in Boolean function f).	$2m(n-1) + n$, where m is the number of terms in function f and n is the number of literals in function f .
Minimal-MUMCUT	Syntactic	ENF, TNF, LNF, TOF, ORF., ORF+, LOF, LIF, LRF	CACC, GACC, MUTP, MUTP/NFP, MUFP, MUFP/NTP	MUMCUT, MAX-A, MAX-B	$m + 1$ to $2m + 1$ where m is the number of terms	Uncertain, but less than $2m(n-1) + \frac{mn^2}{2}$ where m is the number of terms and n is the number of literals
MUTP / MNFP / CUTPNFP Strategy (MUMCUT)	Syntactic	ENF, TNF, LNF, TOF, ORF, LOF, LIF, LRF	RACC, CACC, GACC, MUTP, MUFP, UTPC, CUTPNFP, PCUTPNFP, MUTP/NFP, MNFP, MNTP, CUFPNTP, PCUFPNTP, MUFP/NTP, Minimal-MUMCUT	MAX-A, MAX-B	When infeasibilities arise – m to $2m + 1$ (where m is the number of terms in Boolean function f). Uncertain otherwise.	$2m(n-1) + \frac{mn^2}{2}$, where m is the number of terms in function f and n is the number of literals in function f .

* When feasible, detects the LRF

** When feasible, detects any LRF that the MUFP criterion will not detect

The complementary relationship between minimal CNF and minimal DNF exists throughout the nine single fault types in Lau and Yu's fault hierarchy and thus a new minimal CNF Fault Hierarchy is presented in Figure 8.

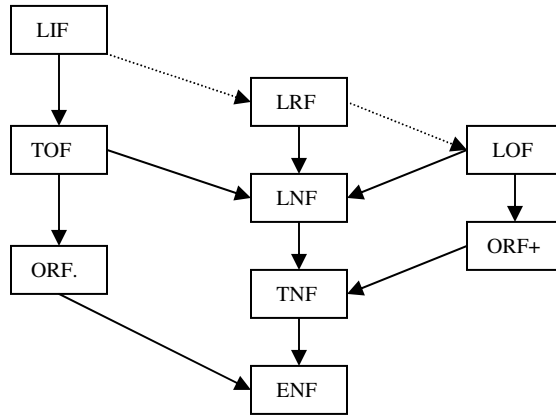


Figure 8 Minimal CNF Fault Hierarchy [22]

The minimal CNF fault hierarchy is identical to the minimal DNF fault hierarchy except that the ORF. and ORF+ have switched. This is because in minimal DNF, the OR operator separates terms and the AND operator separates literals, while in minimal CNF, the AND operator separates terms and the OR operator separates literals. The minimal DNF and minimal CNF fault hierarchies are each composed of three columns. Table 18 shows how the faults in each column of Lau and Yu’s fault hierarchy (minimal DNF fault hierarchy) are related to UTPs and NFPs. Table 19 shows how the faults in each column of Figure 8 (minimal CNF fault hierarchy) are related to UFPs and NTPs. In these tables, an “X” means that the faults in the column header can be detected by the type of point in the row header.

Table 18 Minimal DNF Fault Detection

	LIF,TOF,ORF+	LRF,LNF, TNF, ENF	LOF, ORF.
UTP	X		
UTP or NFP		X	
NFP			X

Table 19 Minimal CNF Fault Detection

	LIF, TOF, ORF.	LRF,LNF, TNF, ENF	LOF, ORF+
UFP	X		
UFP or NTP		X	
NTP			X

For a minimal DNF predicate, a fault in the first column can be detected only by a UTP. For a minimal CNF predicate, a fault in the first column can be detected only by a UFP. For a minimal DNF predicate, a fault in the second column can be detected by a UTP or an NFP. For a minimal CNF predicate, a fault in the second column can be detected by a UFP or an NTP. For a minimal DNF predicate, a fault in the third column can be detected only by an NFP. For a minimal CNF predicate, a fault in the third column can be detected only by an NTP. There is a complementary relationship between minimal DNF and CNF in terms of their logic coverage criteria. MUTP complements MUFP, PCUTPNFP complements PCUFPNTP and MNFP complements MNTP. The Minimal-MUMCUT algorithm can be modified so that it can accept either a minimal CNF or minimal DNF expression without sacrificing fault detection. This is best seen with an example.

Consider $ab + cd$, which is in minimal DNF. The MUTP criterion is feasible for each term, so only a single NFP is needed for each literal and NFPs can be chosen to overlap to reduce test set size. The Minimal-MUMCUT algorithm will generate six tests:

1101 – UTP for ab

1110 – UTP for ab

0111 – UTP for cd

1011 – UTP for cd

0101 – NFP for a and c

1010 – NFP for b and d

Consider $(a + c)(b + d)(a + d)(b + c)$, which is in minimal CNF. With the Minimal-MUMCUT algorithm described so far, this expression would need to first be transformed to minimal DNF before tests are created. Transforming to minimal DNF yields $ab + cd$, for which Minimal-MUMCUT requires the six tests described above.

Now consider a TOF in $(a + c)(b + d)(a + d)(b + c)$ where term $(a + d)$ is omitted to yield $(a + c)(b + d)(b + c)$. This fault can only be detected by 0110 as this point causes the original expression to be FALSE and the faulty expression to be TRUE since 0110 is the lone UFP for term $(a + d)$. Also, consider a TOF in $(a + c)(b + d)(a + d)(b + c)$ where term $(b + c)$ is omitted to yield $(a + c)(b + d)(a + d)$. This fault can only be detected by 1001 as this point causes the original expression to be FALSE and the faulty expression to be TRUE since 1001 is the lone UFP for term $(b + c)$.

Neither 0110 nor 1001 is in the Minimal-MUMCUT test set for $ab + cd$. Although these points could have been chosen for a Minimal-MUMCUT test set they were not because NFPs were chosen to overlap to reduce test set size. While this does not cause any decrease in fault detection if the predicate is in minimal DNF, it does if the predicate is in minimal CNF and then is converted to minimal DNF.

General Form Boolean Expressions

Based on the complementary relationship of minimal DNF and minimal CNF, the Minimal-MUMCUT algorithm can be modified to handle minimal CNF and minimal DNF predicates and still guarantee fault detection. Even better, the algorithm can be modified to handle predicates that are not in minimal CNF or minimal DNF and still have excellent (but not guaranteed) logic fault detection. The modified Minimal-MUMCUT algorithm for general form Boolean expressions is given below and results in a new criterion called Union Minimal-MUMCUT.

Union Minimal-MUMCUT Test Generation Algorithm for General Form Expressions [22]

```
if the expression is in Minimal DNF
  for each term X
    generate MUTP tests for X
    if the MUTP criterion is infeasible for X
      for each literal x in X
        generate PCUTPNFP tests for x
        if the PCUTPNFP criterion is infeasible for x, generate MNFP tests for x
      end for
    else generate an NFP for each literal x in X to overlap NFPs
  end for
else if the expression is in Minimal CNF
  for each term X
    generate MUFP tests for X
    if the MUFP criterion is infeasible for X
      for each literal x in X
        generate PCUFPNTP tests for x
        if the PCUFPNTP criterion is infeasible for x, generate MNTP tests for x
      end for
    else generate an NTP for each literal x in X to overlap NTPs
  end for
else
  convert the expression to Minimal DNF and form tests
  convert the expression to Minimal CNF and form tests
  generate the union of the two test sets known as a Union Minimal-MUMCUT test set
```


The new algorithm for $(a + c)(b + d)(a + d)(b + c)$ would generate tests as follows.

0101 – UFP for $(a + c)$
1010 – UFP for $(b + d)$
0110 – UFP for $(a + d)$
1001 – UFP for $(b + c)$

The MUFP criterion is infeasible for each term, so tests needed to satisfy PCUFPNTP are

1101 – corresponding NTP for a in $(a + c)$ and b in $(b + c)$
0111 – corresponding NTP for c in $(a + c)$ and d in $(a + d)$
1110 – corresponding NTP for b in $(b + d)$ and a in $(a + d)$
1011 – corresponding NTP for d in $(b + d)$ and c in $(b + c)$

Note now that 0101 and 1010 are (and must be) included, which guarantees fault detection of the TOFs in $(a + c)(b + d)(a + d)(b + c)$ examined previously. The modified Minimal-MUMCUT algorithm now guarantees fault detection for the single faults in Lau and Yu's fault hierarchy if the expression under test is in minimal DNF or minimal CNF.

This leaves the case where the predicate is neither in minimal DNF nor minimal CNF. In this case, the expression is converted to each format, tests are generated based on each format and then the union of the two test sets is produced. The union of the two test sets is called a Union Minimal-MUMCUT test set. While such a test set does not guarantee fault detection, it can detect faults that neither the minimal DNF nor minimal CNF test set can with little increase in test set size [22].

A study using 10 predicates from safety critical software was performed to see if Minimal-MUMCUT tests detect a high percentage of faults in predicates that are neither in minimal DNF nor minimal CNF. For each predicate, each possible fault amongst the fault types in Lau and Yu's fault hierarchy was generated. In addition, the following fault

types were generated: Stuck-At Fault (SAF – setting a literal to TRUE or FALSE), Parentheses Omission Fault (POF – omitting a set of parentheses), Parentheses Insertion Fault (PIF – inserting a set of parentheses), Associative Shift Fault (ASF – changing term associativity by moving a set of parentheses). A determination of equivalency for each seeded fault was made by determining if a test set satisfying combinatorial coverage could detect the fault. Any fault for which a combinatorial test set could not detect the fault was determined to be equivalent. A total of 3974 faults were seeded, but 404 (10%) were equivalent, leaving 3570 non-equivalent seeded faults. The predicates in general form are in Appendix C along with examples of the SAF, POF, PIF and ASF.

When a predicate is not in minimal DNF or minimal CNF, the concept of a term does not apply, which affects how a TNF, TOF and LIF are conceptualized. For this study, a term for a predicate neither in minimal DNF nor minimal CNF is either a quantity contained in parentheses or an operand of an OR operator. For example, $a(\sim b + \sim c)d + e$ has five terms: $\sim b$, $\sim c$, $(\sim b + \sim c)$, e and $a(\sim b + \sim c)d$.

Table 20 displays the results of the study showing fault detection for three different Minimal-MUMCUT test sets:

- 1) a test set based only on the minimal DNF form of the predicate
- 2) a test set based only on the minimal CNF form of the predicate
- 3) a Union Minimal-MUMCUT test set

Table 20 Fault Detection in General Form Boolean Predicates by a Minimal-MUMCUT Test Set based on Minimal DNF, Minimal CNF, or Union [22]

No.	Number of Faults	Number of Faults Detected by DNF tests	Percent Detected by DNF tests	Number of Faults Detected by CNF tests	Percent Detected by CNF tests	Number of Faults Detected by Union tests	Percent Detected by Union tests
1	496	496	100%	374	75%	496	100%
4	103	103	100%	102	99%	103	100%
6	859	816	95%	807	94%	816	95%
8	446	437	98%	427	96%	437	98%
9	199	199	100%	183	92%	199	100%
10	367	361	98%	297	81%	365	99%
13	419	381	91%	419	100%	419	100%
14	283	280	99%	237	84%	280	99%
19	237	237	100%	104	44%	237	100%
20	161	161	100%	91	57%	161	100%
Sum	3570	3471	97%	3041	85%	3513	98%

The results show that a Union Minimal-MUMCUT test set detected over 98% of the faults. One interesting finding is that the difference in fault detection for a Union Minimal-MUMCUT test set versus a minimal DNF Minimal-MUMCUT test set was very small (98% versus 97%) whereas the difference in fault detection for a Union Minimal-MUMCUT test set versus a minimal CNF Minimal-MUMCUT test set was larger (98% versus 85%). This occurred even though the difference in test set size of the minimal CNF Minimal-MUMCUT test set and the minimal DNF Minimal-MUMCUT test set was less than 1 test on average. Also, the variability in fault detection was larger for a minimal CNF Minimal-MUMCUT test set than for a minimal DNF Minimal-MUMCUT test set. With different predicates however, this finding could be reversed – meaning that

fault detection based on a minimal DNF Minimal-MUMCUT test set could be lower than the fault detection based on a minimal CNF Minimal-MUMCUT test set. This shows the importance of constructing a Union Minimal-MUMCUT test set to increase fault detection because such a test set will always detect at least as many faults as a Minimal-MUMCUT test set based only on minimal DNF or minimal CNF.

5 Comparison of Minimal-MUMCUT with MUMCUT

This chapter is the first of two chapters comparing the Minimal-MUMCUT criterion with another logic coverage criterion. The focus of this chapter is comparing the Minimal-MUMCUT criterion with the MUMCUT criterion, which corresponds to the first row in Table 5. Each section in this chapter corresponds to a cell in Table 5. Section 5.1 corresponds to cell 1a (test set size comparison), section 5.2 corresponds to cell 1c (single minimal DNF fault detection comparison), section 5.3 corresponds to cell 1d (double minimal DNF fault detection comparison), and section 5.4 corresponds to cell 1e (general logic fault detection comparison). The overriding theme of this chapter is that single and double fault detection of the Minimal-MUMCUT and MUMCUT criteria are identical for minimal DNF/CNF predicates (and very similar for predicates that are neither in minimal DNF nor minimal CNF) yet Minimal-MUMCUT test set size is smaller.

5.1 Test Set Size (Contribution 1a Parts I and II)

Test Set Size in the Minimal DNF Domain

With respect to Lau and Yu's fault hierarchy, a MUMCUT test set may not be minimal because it is possible that tests can be removed from the test set without sacrificing fault detection. From Figure 4 and the Minimal-MUMCUT Test Generation Algorithm in section 4.1, it should be clear that Minimal-MUMCUT test set size will

always be less than or equal to MUMCUT test set size. While MUMCUT always satisfies MUTP, CUPTNFP and MNFP, Minimal-MUMCUT is only required to satisfy MUTP of these three because it takes advantage of feasibility. Consider the predicate $ab + cd$. 1101 and 1110 are UTPs for ab and the MUTP criterion is feasible for ab . 0101 and 1010 are NFPs for a and b , respectively. 0111 and 1011 are UTPs for cd and the MUTP criterion is feasible for cd . 0101 and 1010 are NFPs for c and d , respectively. A Minimal-MUMCUT test set can thus be achieved by satisfying MUTP/NFP such as {1101, 1110, 0101, 1010, 0111, 1011}. This test set has two fewer tests than the smallest test set that can be used to satisfy MUMCUT. Additional tests of 0110 and 1001 would be needed to satisfy MNFP and hence MUMCUT.

Chen, Lau and Yu [8] evaluated MUMCUT test set size (using the greedy MUTP algorithm developed by Chen and Lau [6]) for 19 minimal DNF predicates from an air traffic collision avoidance system (TCAS). There were actually 20 predicates but number 12 was excluded due to a missing a right parenthesis detected by Weyuker et al. [46]. The predicates have from 5 to 13 unique literals (see Appendix B). Kaminski and Ammann [19] created Minimal-MUMCUT tests for each predicate and assessed MUTP feasibility for each term and PCUTPNFP feasibility for each literal. The Minimal-MUMCUT algorithm presented earlier was implemented in Java and used to obtain the results.

On average, Minimal-MUMCUT test set size was 63.72% of MUMCUT test set size (**Contribution 1a Part I**). The greatest savings was for predicate 19, where Minimal-MUMCUT test set size was 35.87% of MUMCUT test set size. Table 21 displays these results. Minimal-MUMCUT test set size is always less than MUMCUT

test set size, except when each literal is in each term, in which case test set size is the same (see predicates 8 and 9 in Appendix B). In this case, each term has only one UTP, each literal has only one NFP (which happens to be a corresponding NFP) and no LIFs or LRFs exist.

Table 21 Minimal-MUMCUT vs. MUMCUT Test Set Size [19]

Predicate	Minimal-MUMCUT [19]	MUMCUT [50]	Percentage	2^n
1	27	39.0	69.23%	128
2	81	116.00	69.83%	512
3	148	238.7	62.00%	4096
4	9	11.8	76.27%	32
5	34	43.0	79.07%	512
6	62	84.0	73.81%	2048
7	62	106.0	58.49%	1024
8	36	36.00	100.00%	256
9	16	16.00	100.00%	128
10	62	86.0	72.09%	8192
11	61	124.0	49.19%	8192
12	N/A	N/A	N/A	N/A
13	17	36.1	47.09%	4096
14	22	34.0	64.71%	128
15	39	60.7	64.25%	512
16	104	153.1	67.93%	4096
17	39	76.3	51.11%	2048
18	48	78.4	61.22%	1024
19	16	44.6	35.87%	256
20	14	24.0	58.33%	128
Sum	897	1407.7		37,408
Avg	47.21	74.09	63.72%	1968.84

Test Set Size in the General Form Domain

Yu and Lau [48] found that of a sample of 20 non-minimal DNF predicates, 99% of seeded faults were detected by MUMCUT tests based on the corresponding minimal DNF predicates. Sun et al. [41] extended MUMCUT in order to guarantee detecting the 1% of faults that went undetected in these general form predicates. They identified five patterns of faults that MUMCUT tests are not guaranteed to detect when such faults are translated into faults in a minimal DNF predicate. These fault types do not exist in Lau and Yu's fault hierarchy. For two of the five patterns (patterns 3 and 4), Sun et al. state that they could not determine a MUMCUT extension to guarantee fault detection. The author confirmed with Sun et al. that errors in their descriptions of patterns 3 and 4 led to their inability to determine MUMCUT extensions. These errors are described and corrected below. Each of the five patterns is described below. Sun et al. do not provide test set size metrics for the extensions, but Kaminski and Ammann [22] did. These metrics are also presented below.

PATTERN 1

For pattern 1, the example given by Sun et al. is mutating abc to $abc + \sim a \sim b$. Note that 001 detects this fault. However, this fault is not included in a MUMCUT test set. To include such a point, an extension known as n -MNFP for $n > 1$ is needed. n -MNFP means to extend MNFP by applying it to all combinations of literals in a term. Whereas 1-NFP (which is the same as NFP) means that only a single literal in a term needs to be negated for the predicate to change from FALSE to TRUE, n -NFP means that n literals in a term need to be negated for the predicate to change from FALSE to TRUE. Note that 2-

NFP for literals a and b in term abc means that term abc evaluates to FALSE but if literal a and literal b are negated, the term abc evaluates to TRUE. Thus, 001 satisfies 2-NFP for literals a and b in term abc . Other 2-NFP points include 010 for literals a and c in term abc and 100 for literals b and c in term abc . Similarly, 3-NFP is satisfied by the point 000 since all 3 literals need to be negated in term abc to change the predicate value from FALSE to TRUE for this point. n-MNFP test set size is based on combinations. In a minimal DNF predicate, either each term contains all unique literals or each term does not. For example, in $ab + bc$ neither term contains all unique literals, but in $ab + \sim a \sim b$ each term contains all unique literals. Based on these observations, n-MNFP size for $n > 1$ is as follows. Let m be the number of terms and let n_i be the number of literals in term i . If each term contains all unique literals then n-MNFP size for $n > 1$ will consist of all FALSE points except for 1-NFPs. If each term does not contain all unique literals, maximum n-MNFP test set size can be at least $2 \sum_{i=1}^m \sum_{r=2}^{n_i} n_i! / r!(n_i - r)!$. That is, for each term, all combinations of “ n_i choose r ” for $r > 1$. The multiplication by two occurs because it takes at least two tests for each n-NFP to satisfy n-MNFP as each literal not in the term of interest must attain values 1 and 0. However, n-MNFP test set size in practice will often be less because some of the combinations of values that make one term FALSE will either (1) represent a TRUE point for another term, (2) represent a 1-NFP for a literal in another term, or (3) represent another n-NFP for a literal in another term. For example, consider $ab + \sim a \sim b$. While 00 is a FALSE point for term ab , it is not a 2-NFP for either literal a or literal b in term ab because this point makes term $\sim a \sim b$ TRUE. As another

example, consider $abc + \sim a \sim b \sim c$. While 001 is a FALSE point for term abc it is not counted as a 2-NFP for either literal a or literal b in term abc because it is a 1-NFP for literal c in term $\sim a \sim b \sim c$.

PATTERN 2

For pattern 2, the example from Sun et al. is mutating $ab + c$ to $ab + bc + ac$. To detect this fault, 001 must be included. While this is a UTP for term c , it does not have to be included in a MUMCUT test suite because 101 and 011 satisfy MUTP for term c . Thus, the extension for this pattern is to include all UTPs. Let m be the number of terms and n be the number of unique literals and n_i be the number of literals in term i . Maximum test set size to include all UTPs is $\sum_{i=1}^m 2^{n-n_i}$. That is, every possible combination of literals not

in the term must be included for each UTP for each term. When the MUTP criterion is infeasible or when OTPs exist, test set size will be less.

PATTERN 3

For pattern 3, Sun et al. give the example of mutating $acde + bc$ to $acde + bc + ab \sim d$. They also state that 11001 is not an NFP for any literal in $acde + bc$. However, 11001 is an NFP for literal c in term bc . Sun et al. state that the fault where $ab \sim d$ is inserted into $acde + bc$ is guaranteed not to be detected by a MUMCUT test suite. However, 11001 can detect the fault as $acde + bc$ evaluates to FALSE but $acde + bc + ab \sim d$ evaluates to TRUE for this point. 11000 is also an NFP for literal c in term bc but this point does not detect the fault. Due to MUMCUT's non-deterministic nature, either 11001 or 11000 could be selected as the NFP for literal c in term bc . Thus, MUMCUT may or may not

detect this fault. Thus, the extension to MUMCUT needed for pattern 3 is to include all NFPs. Let m be the number of terms and n_i be the number of literals in term i . Maximum test set size to include all NFPs is $\sum_{i=1}^m n_i 2^{n-n_i}$. That is, for each literal in each term, all possible combinations of all literals not in the term must be achieved. When infeasibilities arise or when certain combination of literals not in the term make some other term evaluate to TRUE, test set size will be less.

PATTERN 4

For pattern 4, Sun et al. give the example of mutating $ab + ac$ to $ab\sim c + a\sim bc + \sim abc$. 011 can detect this fault as $ab + ac$ evaluates to FALSE but $ab\sim c + a\sim bc + \sim abc$ evaluates to TRUE. 011 must be chosen in a MUMCUT test suite because this point is needed to satisfy MNFP for literal a in term ab and to satisfy MNFP for literal a in term ac . Sun et al. incorrectly claim that only the point 111 can detect the fault and since this point is for certain not chosen in a MUMCUT test suite, they also incorrectly claim that MUMCUT is guaranteed to miss detecting this fault. Upon discussion with Sun et al., the example should have been mutating $ab + ac + bc$ to $ab\sim c + a\sim bc + \sim abc$. Using this example, their analysis is correct in that only 111 can detect the fault. 111 is an OTP as it makes terms ab , ac and bc all TRUE. The extension to MUMCUT then is to include an OTP for every combination of two terms. (In the example above, the OTP for terms ab and ac also happens to be an OTP for terms ab and bc and for terms ac and bc , but this overlap does not necessarily happen.) Let m be the number of terms. Maximum test set size to include an OTP for each combination of two terms is $m! / 2!(m-2)!$. That is, all

combinations of “ m choose 2”. When infeasibilities arise or overlap can occur amongst OTPs, test set size will be less.

PATTERN 5

For pattern 5, Sun et al. give the example of mutating ab to $ab + bc$. No test in a MUMCUT test set will detect this fault because a new literal c is introduced. Consider the general form Boolean expression $(a + b)(c + b) + bd$ and the mutation where the second occurrence of literal b is replaced by literal d to yield $(a + b)(c + d) + bd$. Transforming the original predicate to minimal DNF yields $ac + b$ so the fault goes undetected as variable d no longer appears. The extension Sun et al. give is to include all combinations of missing variables when creating mutants results in the number of unique literals changing between the original predicate and the mutant. Thus, the extension will increase MUMCUT size by a factor of 2^x where x is the number of missing variables.

To summarize, the following extensions to MUMCUT are proposed for general form predicates.

1. Include n-MNFP for $n > 1$
2. Include all UTPs
3. Include all NFPs
4. Include an OTP for each two term combination
5. Include all combinations of missing variables

While this approach increases fault detection in general form predicates, it increases test set size. Test set sizes given previously for each of the five patterns grow very large as the number of literals, number of unique literals and number of terms

grows. Furthermore, none of the points needed for any pattern overlap with any of the points needed for any other pattern. For example, a UTP does not overlap with any n-NFP or any OTP. Including all of these tests can approach or even demand exhaustive coverage as demonstrated by Kaminski and Ammann [22].

An empirical study was conducted by Kaminski and Ammann [22] using 10 of the predicates in Appendix B. Each predicate is converted to minimal DNF and minimal CNF. A minimal DNF and a minimal CNF test set are constructed using the Minimal-MUMCUT algorithm. A MUMCUT extension test set is also constructed. The relative test set sizes of the minimal DNF test set, the minimal CNF test set, the union test set, the MUMCUT test set and the MUMCUT extension test set are then compared. When considering MUMCUT extensions the missing variable case for pattern 5 is not included so as to restrict attention to faults where the same number of unique literals occurs in the original and faulty predicate. Obviously, including tests for all x missing unique literals will add an additional 2^x tests to the MUMCUT extension test set and not including any of these tests in a Minimal-MUMCUT test set will mean that any fault involving the missing unique variable may be missed. Since these are known results, only patterns 1 through 4 are considered. The results are displayed in Table 22.

Table 22 Minimal-MUMCUT, MUMCUT and MUMCUT Extension Test Set Sizes [22]

No.	Minimal-MUMCUT DNF [22]	Minimal-MUMCUT CNF [22]	Union Minimal-MUMCUT [22]	MUMCUT [50]	MUMCUT Extension
1	27	20	30	39.0	128
4	9	10	12	11.8	29
6	62	37	65	84.0	2046
8	36	26	36	36.00	256
9	16	14	16	16.00	128
10	62	22	80	86.0	8186
13	17	88	88	36.1	1874
14	22	22	39	34.0	127
19	16	22	30	44.6	256
20	14	14	18	24.0	128
Sum	281	275	414	411.5	13158

The results show that when the predicates are in neither minimal DNF nor minimal CNF, Union Minimal-MUMCUT test set size is 3.15% of MUMCUT extension test set size (**Contribution 1a Part II**). When a predicate is in neither minimal CNF nor minimal DNF the Minimal-MUMCUT algorithm will convert the predicate to each format and then union the two test sets. The results also show that the MUMCUT extension approach requires on average 85% of the tests needed for exhaustive coverage and for 5 of the 10 predicates exhaustive coverage was actually required. If none of the 10 predicates are in minimal DNF or minimal CNF, test set size is still just 3% of exhaustive size using the Minimal-MUMCUT approach to form a union test set. The actual Minimal-MUMCUT and MUMCUT Extension test sets for four of the predicates in the study are given in Appendix D.

5.2 Single Minimal DNF Fault Detection (Contribution 1c)

This section presents a theoretical contribution by proving the single Minimal DNF fault detection capability of a Minimal-MUMCUT test set and a MUMCUT test set.

Theorem 1 (Contribution 1c): Minimal-MUMCUT vs. MUMCUT Single Minimal DNF Fault Detection

For minimal DNF predicates, Minimal-MUMCUT tests and MUMCUT tests have the same guaranteed single logic fault detection (9 of 9 fault types in Lau and Yu's fault hierarchy).

Proof:

Chen, Lau and Yu [8] show that MUMCUT tests are guaranteed to detect all single faults in Lau and Yu's fault hierarchy. MUTP is guaranteed to detect the LIF and hence the TOF, ORF+, LNF, TNF and ENF [8] and Minimal-MUMCUT always incorporates MUTP. A single NFP is guaranteed to detect the LOF and ORF+ [8] and Minimal-MUMCUT always incorporates at least a single NFP for each unique literal. When feasible, MUTP detects the LRF [8]. When feasible, PCUTPNFP detects the LRF [22]. When feasible, MNFP detects the LRF [8]. Minimal-MUMCUT always incorporates MUTP and will incorporate PCUTPNFP when MUTP is infeasible, unless PCUTPNFP is also infeasible in which case Minimal-MUMCUT will incorporate MNFP. Thus, the LRF is detected, meaning all single fault types are detected by Minimal-MUMCUT tests.

End Proof

5.3 Double Minimal DNF Fault Detection (Contribution 1d Parts I and II)

This section presents a theoretical contribution by proving the double Minimal DNF fault detection capability of a Minimal-MUMCUT test set and a MUMCUT test set.

Theorem 2 (Contribution 1d Part I): Minimal-MUMCUT vs. MUMCUT Double Minimal DNF Fault Detection

For minimal DNF predicates, Minimal-MUMCUT and MUMCUT tests have the same guaranteed double logic fault detection (84 of 92 double fault types in Lau and Yu's fault hierarchy).

Proof:

The Minimal-MUMCUT and MUMCUT criteria subsume the BASIC criterion, which Lau et al. [27, 28, 29] prove detects 84 of 92 double fault types in Lau and Yu's fault hierarchy. However, neither subsumes the additional criteria proposed by Lau et al. needed to guarantee detection of all double fault types. Thus, Minimal-MUMCUT and MUMCUT test sets guarantee the same double fault detection.

End Proof

Both Minimal-MUMCUT and MUMCUT test sets can fail to detect eight double fault types [20]. As was described in section 4.4, the only one of these eight double fault types that went undetected by Minimal-MUMCUT tests in 19 examined predicates is the LIF-LIF as the other seven double fault types were guaranteed to be detected. This same double fault goes undetected by MUMCUT tests because MUMCUT does not require OTPs, which are necessary to detect the LIF-LIF when both terms involved in the double fault are MUTP infeasible [20]. Table 13 showed that over 99% of the LIF-LIFs were

actually detected by Minimal-MUMCUT tests. As an additional component of this empirical study, MUMCUT tests were generated and these tests detected the exact same percentage of the LIF-LIFs as the Minimal-MUMCUT tests did (**Contribution 1d Part II**).

Table 23 compares test set size for the Minimal-MUMCUT criterion supplemented with tests needed to detect LIF-LIFs with mean MUMCUT test set size. The number of tests needed to detect LIF-LIFs is less than the number of undetected LIF-LIFs because multiple undetected LIF-LIFs can be detected by the same test. 129 LIF-LIFs (out of 149,819) went undetected (an average of 6.79 per predicate) but 108 additional tests can be used to detect them (an average of 5.68 tests per predicate). On average, the test set size formed by combining Minimal-MUMCUT tests and tests to detect all LIF-LIFs is 71.39% of MUMCUT test set size (and 2.60% of exhaustive test set size), yet the former detected all faults the MUMCUT tests did plus the LIF-LIF that the MUMCUT tests did not.

Table 23 Minimal-MUMCUT + LIF-LIF Test Set Size vs. MUMCUT Test Set Size [20]

Predicate	Minimal – MUMCUT + LIF-LIF Tests [20]	MUMCUT [50]	Percentage	2^n
1	29	39.0	74.36%	128
2	84	116.00	72.41%	512
3	173	238.7	72.48%	4096
4	10	11.8	84.75%	32
5	41	43.0	95.35%	512
6	64	84.0	76.19%	2048

Predicate	Minimal – MUMCUT + LIF-LIF Tests [20]	MUMCUT [50]	Percentage	2 ⁿ
7	62	106.0	58.49%	1024
8	36	36.00	100.00%	256
9	16	16.00	100.00%	128
10	68	86.0	79.07%	8192
11	64	124.0	51.61%	8192
12	N/A	N/A	N/A	N/A
13	21	36.1	58.17%	4096
14	27	34.0	79.41%	128
15	56	60.7	92.26%	512
16	133	153.1	86.87%	4096
17	39	76.3	51.11%	2048
18	52	78.4	66.33%	1024
19	16	44.6	35.87%	256
20	14	24.0	58.33%	128
Sum	1005	1407.7		37,408
Avg	52.89	74.09	71.39%	1968.84

5.4 General Logic Fault Detection (Contribution 1e Parts I and II)

In section 4.5 it was reported that of 3% of 20,256 predicates Chilenski extracted from avionics software contained five or more unique literals, and that 85% of these predicates were either in minimal DNF or minimal CNF (or both). Section 4.5 also showed that the Union Minimal-MUMCUT approach guarantees fault detection of all single and 84 of 92 double faults in Lau and Yu's fault hierarchy for predicates in either minimal DNF or minimal CNF.

If a minimal CNF predicate is converted to minimal DNF, and a MUMCUT test set is formed, that test set will satisfy MUTP, CUTPNFP, and MNFP for the minimal

DNF predicate, but it will also satisfy MUFP, CUFNTP, and MNTP for the original minimal CNF predicate. This is due to the dual nature of minimal CNF and minimal DNF and the correspondence between MUTP and MUFP, CUTPNFP and CUFNTP, and MNFP and MNTP. Thus, for 85% of the predicates in the study, fault detection was not compromised for either MUMCUT tests or Union Minimal-MUMCUT tests **(Contribution 1e Part I)**.

Although MUMCUT fault detection has been shown to be very good for general form predicates, the question remains as to how well Minimal-MUMCUT tests do at fault detection for general form predicates. Yu and Lau [48] found that of a sample of 20 predicates that are neither in minimal DNF nor minimal CNF, 99% of seeded faults were detected by MUMCUT tests formed from the minimal DNF version of the predicates. 10 of these 20 predicates were the exact same predicates used in the empirical study described in section 4.5 that showed that Union Minimal-MUMCUT tests detected over 98% of seeded faults. Also, the fault types matched between the two studies. Thus, a direct comparison for these 10 predicates can be done. The MUMCUT tests generated by Yu and Lau for these 10 predicates also detected over 98% of the faults **(Contribution 1e Part II)**. Thus, the results indicate that the ability of a Union Minimal-MUMCUT test set to detect faults in general form expressions is comparable to the ability of a MUMCUT test set despite the smaller Union Minimal-MUMCUT test set size.

6 Comparison of Minimal-MUMCUT with RACC and RICC

This chapter is the second of two comparing the Minimal-MUMCUT criterion with another logic coverage criterion. The focus of this chapter is comparing the Minimal-MUMCUT criterion with RACC and RICC, which corresponds to the second row in Table 5. Each section in this chapter corresponds to a cell in Table 5. Section 6.1 corresponds to cell 2a (test set size comparison), section 6.2 corresponds to cell 2c (single minimal DNF fault detection comparison), and section 6.3 corresponds to cell 2d (double minimal DNF fault detection comparison). The overriding theme of this chapter is that while the Minimal-MUMCUT criterion has a larger test set size, this is offset by the fact that fault detection for RACC and RICC is worse.

6.1 Test Set Size (Contribution 2a)

Chilenski and Miller [10] state that minimum test set size for both Weak and Strong MCDC is $n + 1$, where n is the number of unique literals for Weak MCDC and the number of literals for Strong MCDC. Chilenski and Miller [10] also state that for both Weak and Strong MCDC, test set size can exceed $n+1$ where n is the number of unique literals for Weak MCDC and the number of literals for Strong MCDC. Finally, Chilenski and Miller [10] also state that a test set size of $2n$ will always suffice for both Weak and Strong MCDC, again where n is the number of unique literals for Weak MCDC and the number of literals for Strong MCDC. However, Chilenski and Miller [10] do not state

that $2n$ tests will actually ever be required (except for $n = 1$) so all that can be concluded from their research is that maximum MCDC test set size is between $n + 2$ and $2n$, inclusive. Ammann and Offutt [2] indicate that RACC and CACC test set size is always $n+1$ where n is the number of unique literals. This conflicts with the claim of Chilenski and Miller [10] since RACC and CACC are versions of weak MCDC. To resolve this discrepancy and to better quantify the range of maximum MCDC test set size, the author analyzed RACC test set size for a large set of predicates.

There exist 2^{2^n} possible Boolean predicates in n unique literals. The author examined RACC test set size for all such predicates for $n=1$, $n=2$ and $n=3$ and found that a RACC test set can always be formed by $n + 1$ tests. Thus, maximum RACC test set size for predicates with 1, 2, or 3 unique literals is $n + 1$. For predicates with 4 unique literals, 65,536 predicates are possible so the author did not examine each of these. However, the author conjectures that when $n=4$, maximum test set size remains $n+1$. When $n \geq 5$, the situation changes as demonstrated by Kaminski and Ammann [21]. Kaminski and Ammann developed the following algorithm for determining the lowest bound on maximum test set size for RACC. This same algorithm applies to CACC.

Lowest Bound Maximum RACC Test Set Size Algorithm [21]

```

if  $n=1$  or  $n=2$  or  $n=3$  or  $n=4^*$ 
    maximum RACC test set size is  $n+1$ 
else if  $n=5$ 
    maximum RACC test set size is at least  $n + 2$ 
else if an integer  $y$  exists such that  $n - (y + 2^y) = 1$ 
    maximum RACC test set size is at least  $2(n-x) - 1$  where  $x$  is the greatest integer such
    that  $x + 2^x \leq n$ 
else
    maximum RACC test set size is at least  $2(n-x)$  where  $x$  is the smallest integer such that
     $x + 2^x \geq n$ 

```

*For $n=4$, this is a conjecture by the author

As an example, consider $n=7$. An integer $y=2$ exists such that $n - (y + 2^y) = 1$.

The greatest integer x such that $x + 2^x \leq n$ holds is $x=2$. Thus, maximum RACC test set size is at least $2(n-x) - 1 = 2(7-2) - 1 = 9$. As another example, consider $n=11$. No integer y exists such that $n - (y + 2^y) = 1$. The smallest integer x such that $x + 2^x \geq n$ holds is $x=3$. Thus, maximum RACC test set size is at least $2(n-x) = 2(11-3) = 16$.

Table 24 displays the results of the maximum RACC test set size for $n=1$ to $n=37$. Note that after $n=37$ the pattern repeats because $5 + 2^5 = 37$.

Table 24 Lowest Bound Maximum RACC Test Set Size Algorithm Results [21]

Number of Unique Literals (n)	Formula	Size	Raw Size
1	$n+1$	$n+1$	2
2	$n+1$	$n+1$	3
3	$n+1$	$n+1$	4
4	$n+1$	$n+1$	5
5	$n+2$	$n+2$	7
6	$2(n-2)$	$n+2$	8
7	$2(n-2) - 1$	$n+2$	9
8	$2(n-3)$	$n+2$	10
9	$2(n-3)$	$n+3$	12
10	$2(n-3)$	$n+4$	14
11	$2(n-3)$	$n+5$	16
12	$2(n-3) - 1$	$n+5$	17
13	$2(n-4)$	$n+5$	18
14	$2(n-4)$	$n+6$	20
15	$2(n-4)$	$n+7$	22
16	$2(n-4)$	$n+8$	24
17	$2(n-4)$	$n+9$	26
18	$2(n-4)$	$n+10$	28
19	$2(n-4)$	$n+11$	30
20	$2(n-4)$	$n+12$	32

Number of Unique Literals (n)	Formula	Size	Raw Size
21	$2(n-4) - 1$	$n+12$	33
22	$2(n-5)$	$n+12$	34
23	$2(n-5)$	$n+13$	36
24	$2(n-5)$	$n+14$	38
25	$2(n-5)$	$n+15$	40
26	$2(n-5)$	$n+16$	42
27	$2(n-5)$	$n+17$	44
28	$2(n-5)$	$n+18$	46
29	$2(n-5)$	$n+19$	48
30	$2(n-5)$	$n+20$	50
31	$2(n-5)$	$n+21$	52
32	$2(n-5)$	$n+22$	54
33	$2(n-5)$	$n+23$	56
34	$2(n-5)$	$n+24$	58
35	$2(n-5)$	$n+25$	60
36	$2(n-5)$	$n+26$	62
37	$2(n-5)$	$n+27$	64

Note from Table 24 that for $n=37$, RACC test set size can be at least 64, which is $1.73n$. For $n=1034$, RACC test set size can be at least 2048 or $1.98n$ according to the algorithm. While it is doubtful that any predicate in practice contains 1034 unique literals, from a theoretical perspective as n approaches infinity maximum RACC test set size approaches $2n$.

For an example of RACC test set size of $n+5 = 2(n-3) = 16$ for $n=11$ consider

$$abcd + !abce + a!bcf + ab!cg + !a!bch + !a!b!ci + a!b!cj + !ab!ck$$

RACC selects a corresponding UTP-NFP pair for each unique literal (as opposed to each literal) when the predicate is transformed into minimal DNF as UTPs and NFPs translate to the conditions under which each literal determines a predicate. Since literals d, e, f, g,

h, i, j and k each appear in a different term, 8 UTPs are needed for RACC. Note also that the NFPs amongst literals d, e, f, g, h, i, j and k cannot overlap with each other because:

the NFP for literal d requires $a=1, b=1, c=1$

the NFP for literal e requires $a=0, b=1, c=1$

the NFP for literal f requires $a=1, b=0, c=1$

the NFP for literal g requires $a=1, b=1, c=0$

the NFP for literal h requires $a=0, b=0, c=1$

the NFP for literal i requires $a=0, b=0, c=0$

the NFP for literal j requires $a=1, b=0, c=0$

the NFP for literal k requires $a=0, b=1, c=0$

Thus, at this point 16 tests are needed (8 UTPs and 8 NFPs) to satisfy RACC. No additional tests are needed because the NFPs for literals a, b and c can overlap with NFPs for literals d, e, f, g, i, j and k. Note that for $n=11$, $n+5 = 2(n-3)$. Intuitively, $2(n-3)$ tests are needed because all but 3 literals (a, b and c) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a, b and c can overlap with other tests.) RACC test set size analysis for actual predicates with other values of n is given in Appendix E.

To compare RACC and Minimal-MUMCUT test set size in practice, an empirical study was undertaken using the predicates listed in Appendix B. For each predicate, a

RACC test set was constructed manually and a Minimal-MUMCUT test set was generated automatically using the Minimal-MUMCUT algorithm.

Table 25 lists RACC test set size and Minimal-MUMCUT test set size for each predicate. The column labeled 2^n refers to the total number of possible tests where n is the number of unique literals in the predicate. While the author does not guarantee that the RACC test set constructed for each predicate is the smallest possible, the author does guarantee two things. One, the RACC test set for each predicate is minimal in that if any test in the test set is removed, the test set will no longer satisfy RACC. Two, the RACC test set for each predicate is no larger than the lowest bound maximum RACC test set size according to the algorithm presented earlier. Also, a test set size of $n+1$ was the most common test set size amongst the 19 predicates and in no case was a test set size greater than $n+4$ chosen.

Table 25 shows that RACC test set size and Minimal-MUMCUT test set size are both much smaller than combinatorial test set size. RACC test set size is on average just 0.6% of combinatorial test set size and Minimal-MUMCUT test set size is on average just 2.4% of combinatorial test set size. Table 25 also shows that on average RACC test set size is about 25% of Minimal-MUMCUT test set size (**Contribution 2a**).

Table 25 RACC and Minimal-MUMCUT Test Set Size [21]

Predicate	RACC size [21]	Minimal- MUMCUT size [19]	Percentage	2^n
1	9	27	33%	128
2	10	81	12%	512
3	16	148	11%	4096

Predicate	RACC size [21]	Minimal- MUMCUT size [19]	Percentage	2^n
4	6	9	67%	32
5	12	34	35%	512
6	12	62	19%	2048
7	12	62	19%	1024
8	9	36	25%	256
9	8	16	50%	128
10	15	62	24%	8192
11	16	61	26%	8192
12	N/A	N/A	N/A	N/A
13	16	17	94%	4096
14	9	22	41%	128
15	12	39	31%	512
16	16	104	15%	4096
17	14	39	36%	2048
18	14	48	29%	1024
19	10	16	63%	256
20	8	14	57%	128
Average	11.79	47.21		1968.84
Total	224	897	24.97%	37,408

Table 26 displays the average test set size for RACC and Minimal-MUMCUT grouped by the number of unique literals. The table shows that as the number of unique literals increases the trend is that RACC test set size is a smaller percentage of Minimal-MUMCUT test set size. Intuitively, as the number of unique literals increases, there are more opportunities for literals to repeat in different terms in a minimal DNF predicate. As more and more literals repeat in different terms, Minimal-MUMCUT test set size will increase but RACC test set size will not.

Table 26 Average Test Set Size for RACC and Minimal-MUMCUT Grouped by Number of Unique Literals [21]

Number of unique literals	RACC size	Minimal-MUMCUT size	Percentage	2^n
5	6.00	9.00	67%	32
*6	N/A	N/A	N/A	64
7	8.50	19.75	43%	128
8	9.50	26.00	37%	256
9	11.33	52.00	22%	512
10	13.00	57.00	23%	1024
11	13.00	53.00	25%	2048
12	16.00	95.33	17%	4096
13	15.50	67.00	23%	8192

* no predicates examined had 6 unique literals

6.2 Single Minimal DNF Fault Detection (Contribution 2c Parts I and II)

This section presents a theoretical contribution by proving the single Minimal DNF fault detection capability of a RACC test set and a RICC test set.

Theorem 3 (Contribution 2c Part I): Minimal-MUMCUT vs. RACC/RICC Single Minimal DNF Fault Detection

For minimal DNF predicates, a test set that satisfies either RACC or RICC guarantees detecting 2 of the 9 single fault types in Lau and Yu's fault hierarchy (the ENF and TNF).

Proof:

The ENF can be detected by any test [30] and the TNF can be detected by any test for which the predicate evaluates to FALSE [30] and since RACC and RICC are guaranteed to include such tests, these faults are guaranteed to be detected. Examples are shown in

Appendix F of seeding each of the 7 faults into an actual predicate and showing that each test in a RACC test set and each test in a RICC test fails to detect each fault.

End Proof

A natural extension of the work of Chen and Lau [7] is to establish the conditions where the ACC and ICC test series fail to detect the faults in Lau and Yu's fault hierarchy. MCDC (ACC) is widely accepted as the criterion of choice for many software critical applications. However, with respect to the hierarchy in Lau and Yu's fault hierarchy, Kaminski, Williams and Ammann [24] found that tests satisfying a common version of MCDC (RACC) would not detect 7 of the faults under specific circumstances. If the predicate is a singular Boolean expression in minimal DNF, then RACC tests are guaranteed to detect all faults in Lau and Yu's fault hierarchy but the LIF and CACC tests are guaranteed to detect all faults in Lau and Yu's fault hierarchy but the LIF and the LRF [24]. RACC and CACC tests can miss detection of 7 of the 9 faults when literals repeat in terms as shown in Appendix F.

To determine how often RACC (and hence CACC and GACC) tests actually do miss detecting the 7 fault types they are not guaranteed to detect, an empirical study was undertaken using the predicates listed in Appendix B. For each predicate, a RACC test set was constructed manually. Every type of fault was examined manually to determine if the RACC test set could detect it. An example is given in Appendix G.

Table 27 specifies the fault detection capability of the RACC test sets for each predicate. Appendix H specifies the actual RACC tests used for each predicate and the percentage of faults of each fault type that are detected. The results in Table 27 show that,

on average, RACC tests actually detected 35% of the faults Minimal-MUMCUT guarantees detecting. Thus, although RACC test set size is 25% of Minimal-MUMCUT test set size on average, this is offset by the fact that the majority of faults go undetected.

Table 27 RACC Fault Detection [21]

Predicate	Faults RACC detects	Total Faults	Percentage	2^n
1	79	173	46%	128
2	108	548	20%	512
3	583	2493	23%	4096
4	61	71	86%	32
5	267	483	55%	512
6	72	342	21%	2048
7	158	524	30%	1024
8	44	104	42%	256
9	31	46	67%	128
10	268	576	47%	8192
11	267	1047	26%	8192
12	N/A	N/A	N/A	N/A
13	336	397	85%	4096
14	162	236	69%	128
15	275	605	45%	512
16	502	1980	25%	4096
17	240	524	46%	2048
18	274	596	46%	1024
19	108	212	51%	256
20	56	68	82%	128
Average	204.79	580.26		
Sum	3891	11025	35.29%	

Table 28 displays the average fault detection for RACC tests grouped by the number of unique literals. The table shows that as the number of unique literals increases the trend is that RACC tests detect a smaller percentage of faults. Intuitively, as the number of unique literals increases, there are more opportunities for literals to repeat in different terms in the minimal DNF predicate. As more and more literals repeat in different terms RACC test set size will not increase so the additional faults caused by repeated literals have an increased chance of being undetected.

Table 28 RACC Fault Detection Grouped by Number of Unique Literals [21]

Number of unique literals	Faults RACC detects	Total Faults	Percentage	2^n
5	61	71	86%	32
*6	N/A	N/A	N/A	64
7	328	523	63%	128
8	152	316	48%	256
9	650	1636	40%	512
10	432	1120	39%	1024
11	312	956	33%	2048
12	1421	4870	29%	4096
13	535	1623	33%	8192

* no predicates examined had 6 unique literals

RACC tests are guaranteed to detect only the ENF and TNF. Thus, another question is what percentage of the faults that RACC tests may or may not detect do they actually detect? The answer is 34% for the predicates in this study. This is very close to the 35% given in Table 27 because the number of ENFs and TNFs (which RACC tests do guarantee detecting) is a very small percentage of the total number of minimal DNF

faults that can occur in a predicate. That is, only one ENF is possible and the number of TNFs is equal to the number of terms. The other fault types are much more prevalent. Thus, RACC tests missed detecting 66% of the faults that they are not guaranteed to detect (**Contribution 2c Part II**).

The implication of these results is that the extra cost associated with the Minimal-MUMCUT syntactic criterion is justified for safety-critical software since semantic RACC tests missed detecting the majority of faults. For other types of software with large predicates, testers will need to make an informed decision as to which is more important, test set size or fault detection. With RACC, testers can expect to miss 2/3 of the faults in Lau and Yu's fault hierarchy for large predicates with literals that repeat in different terms. With Minimal-MUMCUT, fault detection is guaranteed, but at the cost of a test set size that is likely to be four times as large.

6.3 Double Minimal DNF Fault Detection (Contribution 2d)

This section presents a theoretical contribution by proving the double Minimal DNF fault detection capability of a RACC test set and a RICC test set.

Theorem 4 (Contribution 2d): Minimal-MUMCUT vs. RACC/RICC Double Minimal DNF Fault Detection

For minimal DNF predicates, a test set that satisfies either RACC or RICC guarantees detecting 22 of the 92 double fault types in Lau and Yu's fault hierarchy.

Proof:

RACC and RICC tests require neither a UTP for each term nor an NFP for each literal when a predicate is in minimal DNF (RACC requires a UTP and corresponding NFP for

each *unique* literal). Lau, Liu and Yu [27, 28, 29] document the detection conditions needed for each of the 92 ordered, versioned double fault types. Most of the 92 types require either a UTP or NFP to be detected. Others require TRUE points for detection that although not being UTPs, must satisfy certain conditions such as various literals being TRUE or FALSE. Still other double fault types require FALSE points for detection that although not being NFPs, must also satisfy certain conditions such as various literals being TRUE or FALSE. RACC and RICC do not require these conditions to hold. Of the detection conditions specified, the only kind that RACC and RICC require is that at least one FALSE point be in the test set. 22 of the 92 ordered, versioned double fault types can be detected by an FALSE point. This translates to 4 of the 45 unordered, non-versioned double fault types. Thus RACC and RICC tests guarantee detecting 22 of 92 (24%) of the ordered, versioned double fault types and 4 of the 45 (9%) of the unordered, non-versioned double fault types (ENF-TOF, TNF-TNF, TNF-TOF and TNF-LIF).

End Proof

7 TRF-TIF Logic Mutation

7.1 Overview of TRF-TIF Logic Mutation

Software logic mutation testing can be inefficient for at least three reasons. One, the same logic mutant can be generated multiple times. Two, logic mutants are generated that are guaranteed to be killed by a test that kills some other generated logic mutant. Three, mutation tools lack logic mutation operators that generate mutants which, when killed, guarantee killing the most number of other potential logic mutants. These inefficiencies cause excess mutants to be generated and reduce fault detection capability.

Three new mutation operators are introduced by Kaminski and Ammann [18] to resolve these problems, assuming minimal DNF. These operators are based on three new faults, the Term Reference Fault / Literal Insertion Fault (TRF/LIF), Term Insertion Fault / Literal Reference Fault (TIF/LRF) and Term Insertion Fault / Literal Omission Fault (TIF/LOF). Using these new mutation operators, a smaller mutant test set can be generated yet still detect all LIFs, LRFs and LOFs assuming tests are found to kill the mutants. These new fault types do not exist in Lau and Yu's fault hierarchy and have no corresponding mutation operators in current tools. A TRF/LIF involves replacing a term with one or more terms to guarantee LIF detection. A TIF/LRF involves inserting a single term containing all literals to guarantee LRF detection. A TIF/LOF involves inserting a

single term containing all literals to guarantee LOF detection. The mutation testing approach based on these faults is called TRF-TIF logic mutation.

The rest of this chapter is organized as follows. The remainder of section 7.1 describes the algorithm used to generate TRF-TIF logic mutants. Section 7.1 also introduces an extended fault hierarchy as well as new measures of mutation efficiency. Section 7.2 discusses TRF-TIF logic mutant set size and section 7.3 discusses TRF-TIF equivalent logic mutant set size. Single and double minimal DNF fault detection of a test set weakly killing all TRF-TIF logic mutants is discussed in sections 7.4 and 7.5, respectively. Section 7.6 briefly mentions how the ability of a test set that weakly kills all TRF-TIF logic mutants to kill general mutants is evaluated.

Algorithms are presented below to describe how TRF-TIF logic mutants are generated, starting with TRF/LIF mutations, then proceeding with TIF/LRF mutations, and concluding with TIF/LOF mutations.

TRF/LIF Mutation Algorithm [23]

```

for each term X in the Minimal DNF predicate
  for each non-equivalent LIF that can occur for term X
    create a set of tests that can detect the LIF and mark this test set as unused;
  end for
  while at least one unused LIF test set exists
    if an unused test set contains only one test
      select that test (ties broke arbitrarily);
    else
      select the test that appears in the most unused sets of LIF tests (ties broken
      arbitrarily);
    create a TRF/LIF mutant by replacing term X with a sequence of terms separated by
    OR such that the sequence contains all LIFs that can be detected by the selected test;
    mark any LIF test sets containing the selected test as used;
  end while
end for

```

As an example, consider $ab + cd$. Four non-equivalent LIFs exist for term ab :

Inserting c into ab – detected by any test in test set 1: {1100, 1101}

Inserting $\sim c$ into ab – detected by the lone test in test set 2: {1110}

Inserting d into ab – detected by any test in test set 3: {1100, 1110}

Inserting $\sim d$ into ab – detected by the lone test in test set 4: {1101}

Note that test set 2 contains a single test so this test (1110) is selected. 1110 can detect the LIF where $\sim c$ is inserted into ab and the LIF where d is inserted into ab . Thus, the TRF/LIF mutant replaces ab with $ab\sim c + abd$ to yield $ab\sim c + abd + cd$. Now test sets 2 and 3 are marked as used. Of the remaining unused test sets, test set 4 contains a single test so this test (1101) is selected. 1101 can detect the LIF where c is inserted into ab and the LIF where $\sim d$ is inserted into ab . Thus, the TRF/LIF mutant replaces ab with $abc + ab\sim d$ to yield $abc + ab\sim d + cd$. Now test sets 1 and 4 are marked as used and the algorithm is repeated for term cb .

The number of TRF/LIFs for a term is the number of UTPs needed to make as many external literals (literals not in that term) 0 and 1 as possible. The number of LIFs for a term is twice the number of external literals. Consider $ab + \sim acdefgh$. TRF/LIFs are: $f' = ab\sim c + ab\sim d + ab\sim e + ab\sim f + ab\sim g + ab\sim h + \sim acdefgh$ (whose mutant can only be killed by 11111111) and $f'' = abc + abd + abe + abf + abg + abh + \sim acdefgh$ (whose mutant can only be killed by 11000000). The f' TRF/LIF has a corresponding LIF of $ab\sim c + \sim acdefgh$, whose mutant can be killed by any of 32 inputs: 111XXXXX, where XXXXX is any combination of values for $defgh$. Killing the f' and f'' mutants guarantees killing all 12 LIF mutants for ab . A TRF/LIF has a large syntactic but small semantic

fault size as compared to an LIF. A TRF/LIF involves more syntactic changes than an LIF, yet only one input kills a TRF/LIF mutant whereas several inputs might kill a corresponding LIF mutant. The one input killing a TRF/LIF mutant is the intersection of the input sets that kill each corresponding LIF mutant. The algorithm presented next performs TIF/LRF mutations.

TIF/LRF Mutation Algorithm [23]

```

for each term X in the Minimal DNF predicate
  if an equivalent LIF exists* by inserting some literal y (or its negation) into term X
    for each literal z in term X
      for each non-equivalent LRF where literal z is replaced with y (or its negation)
        create a set of tests that can detect the LRF and mark this test set as unused;
      end for
    end for
  end if
end for
while at least one unused LRF test set exists
  if an unused test set contains only one test
    select that test (ties broke arbitrarily);
  else
    select test that appears in the most unused sets of LRF tests (ties broke arbitrarily);
  create a TIF/LRF mutant by inserting a term containing all literals that evaluates to
  TRUE for the selected test;
  mark any LRF tests containing that test as used;
end while
* There is an exception for a certain special type of equivalent LIF, described below

```

As an example, consider $ab + ac + ad$. Equivalent LIF mutants exist when inserting $\sim c$ or $\sim d$ into ab , when inserting $\sim b$ or $\sim d$ into ac and when inserting $\sim b$ or $\sim c$ into ad . This results in the following non-equivalent LRFs to consider:

Replacing a with $\sim c$ in ab – detected by any test in test set 1: {0100, 0101}

Replacing a with $\sim d$ in ab – detected by any test in test set 2: {0100, 0110}

Replacing b with $\sim c$ in ab – detected by the lone test in test set 3: {1000}

Replacing b with $\sim d$ in ab – detected by the lone test in test set 4: {1000}

Replacing a with $\sim b$ in ac – detected by any test in test set 5: {0010, 0011}

Replacing a with $\sim d$ in ac – detected by any test in test set 6: {0010, 0110}

Replacing c with $\sim b$ in ac – detected by the lone test in test set 7: {1000}

Replacing c with $\sim d$ in ac – detected by the lone test in test set 8: {1000}

Replacing a with $\sim b$ in ad – detected by any test in test set 9: {0001, 0011}

Replacing a with $\sim c$ in ad – detected by any test in test set 10: {0001, 0101}

Replacing d with $\sim b$ in ad – detected by the lone test in test set 11: {1000}

Replacing d with $\sim c$ in ad – detected by the lone test in test set 12: {1000}

Test sets 3, 4, 7, 8, 11 and 12 each contain one test so this test (1000) is selected.

These test sets are marked as used and the corresponding TIF/LRF is created where $a\sim b\sim c\sim d$ is inserted as a new term to yield $ab + ac + ad + a\sim b\sim c\sim d$. Of the remaining tests in the unused test sets, each test occurs twice (for example, 0100 occurs in test set 1 and test set 2). So 0100 is arbitrarily chosen and test sets 1 and 2 are marked as used. The corresponding TIF/LRF is created where $\sim ab\sim c\sim d$ is inserted as a new term to yield $ab + ac + ad + \sim ab\sim c\sim d$. Of the remaining tests in the unused test sets, each test occurs twice (for example, 0010 occurs in test set 5 and test set 6). So 0010 is arbitrarily chosen and test sets 5 and 6 are marked as used. The corresponding TIF/LRF is created where $\sim a\sim bc\sim d$ is inserted as a new term to yield $ab + ac + ad + \sim a\sim bc\sim d$. Of the remaining tests in the unused test sets, 0001 occurs the most times (once in test set 9 and once in test set 10). So 0001 is chosen and test sets 9 and 10 are marked as used. The corresponding

TIF/LRF is created where $\sim a \sim b \sim cd$ is inserted as a new term to yield $ab + ac + ad + \sim a \sim b \sim cd$.

As another example consider $f = ab + ac + ad + ae + fg$. One TIF/LRF will be $f' = ab + ac + ad + ae + fg + \sim ab \sim c \sim d \sim e \sim f \sim g$. Another TIF/LRF will be $f'' = ab + ac + ad + ae + fg + a \sim b \sim c \sim d \sim e \sim f \sim g$. The f' mutant can only be killed by 0100000 and the f'' mutant can only be killed by 1000000. The f' TIF/LRF has a corresponding LRF of $\sim cb + ac + ad + ae + fg$, whose mutant can be killed by any of 12 inputs: 010XXXX, where XXXX is any combination of values for $defg$ such that term fg is FALSE. The f'' TIF/LRF has a corresponding LRF of $a \sim c + ac + ad + ae + fg$, whose mutant can be killed by any of 12 inputs: 100XXXX, where XXXX is any combination of values for $defg$ such that term fg is FALSE. Killing the f' and f'' mutants guarantees killing all six LRF mutants where a or b in ab is replaced with $\sim c$, $\sim d$, or $\sim e$. A TIF/LRF has a large syntactic but small semantic fault size as compared to an LRF. A TIF/LRF involves more syntactic changes than an LRF, yet only one input kills a TIF/LRF mutant whereas several inputs may kill a corresponding LRF mutant. The one input killing a TIF/LRF mutant is the intersection of the input sets that kill each corresponding LRF mutant.

The one exception mentioned in the algorithm is when in each equivalent LIF, the literal being inserted is from a single-literal term. Consider $ab + cd + e$ and note $ab \sim e + cd + e$ is equivalent. However, an LRF where $\sim e$ replaces a results in an LOF for a and is treated as such. When an LRF is equivalent, the corresponding TIF/LRF will not be. Consider $ab + b \sim c + \sim bc$. The LRF $ac + b \sim c + \sim bc$ is equivalent. The TIF/LRF $ab + b \sim c + \sim bc + a \sim bc$ is not produced as it is impossible to make all terms in the original

predicate FALSE when term $a \sim bc$ is TRUE. The algorithm presented next performs TIF/LOF mutations.

TIF/LOF Mutation Algorithm [23]

```

for each term X in the Minimal DNF predicate
  if an equivalent LIF does not exist* when inserting literal y (or its negation) into X
    for each literal z in term X
      for each LOF that occurs by omitting literal z in term X
        create a set of tests that can detect the LOF and mark this test set as unused;
      end for
    end for
  end if
end for
for each LOF test set
  if the LOF test contains a test in any of the tests needed to kill previously generated
  TIF/LRF mutants
    mark the LOF test set as used;
  end for
while at least one unused LOF test set exists
  if an unused test set contains only one test
    select that test (ties broken arbitrarily);
  else
    select test that appears in the most unused sets of LOF tests (ties broken arbitrarily);
  create a TIF/LOF mutant by inserting a term containing all literals that evaluates to
  TRUE for the selected test;
end while
* There is an exception for a certain special type of equivalent LIF, described below

```

As an example, consider $ab + cd$. No equivalent LIFs exist for this predicate so there are 4 LOFs to consider:

Omitting literal a – detected by any test in test set 1: {0100, 0101, 0110}

Omitting literal b – detected by any test in test set 2: {1000, 1001, 1010}

Omitting literal c – detected by any test in test set 3: {0001, 0101, 1001}

Omitting literal d – detected by any test in test set 4: {0010, 0110, 1010}

No TIF/LRF mutants will be generated since no equivalent LIF mutants exist. Thus the algorithm proceeds to the while loop. Tests 1010, 0101, 0110 and 1001 each occur twice amongst the test sets while tests 0100, 1000, 0001 and 0010 each occur once. So any of 1010, 0101, 0110, or 1001 can be chosen and arbitrarily 1010 is chosen and test sets 2 and 4 are marked as used. The corresponding TIF/LOF mutant inserts $a\sim bc\sim d$ as a new term to yield $f' = ab + cd + a\sim bc\sim d$. Of the remaining unused test sets, 0101 is the only test that occurs twice (the other tests in the unused test sets occur only once) so 0101 is selected and test sets 1 and 3 are marked as used. The corresponding TIF/LOF mutant inserts $\sim ab\sim cd$ as a new term to yield $f'' = ab + cd + \sim ab\sim cd$. This algorithm forces the tester to kill the f' mutant with an input (1010) that detects LOFs for b and d and forces the tester to kill the f'' mutant with an input (0101) that detects LOFs for a and c . The f' TIF/LOF has a corresponding LOF (for b) of $a + cd$, whose mutant can be killed by any of 3 inputs: 1000, 1001, 1010. The TIF/LOF mutants can only be killed by NFPs that overlap with other NFPs, increasing the number of corresponding LOF mutants killed by the lone input killing the TIF/LOF mutant. A TIF/LOF has a large syntactic but small semantic fault size as compared to an LOF. A TIF/LOF involves more syntactic changes than an LOF, yet only one input kills a TIF/LOF mutant whereas several inputs may kill a corresponding LOF mutant. The one input killing a TIF/LOF mutant is the intersection of the input sets that kill each corresponding LOF mutant.

The TIF/LOF mutation operator can produce $(n-1) * 2^{n-1}$ fewer mutants than the LOF mutation operator, where n is the number of unique literals. Consider $\sim a\sim b\sim c +$

$ab\sim c + \sim abc + a\sim bc$. There are $n * 2^{n-1} = 12$ LOFs (one per literal) and $2^{n-1} = 4$ TIF/LOFs as only 2^{n-1} FALSE points exist.

The TIF/LOF algorithm is generally restricted to terms with no equivalent LIFs since when an equivalent LIF mutant exists, a corresponding NFP is needed to kill a corresponding LRF mutant. Since any NFP for an omitted literal kills an LOF mutant, generating both a TIF/LOF and TIF/LRF mutant is excessive as the input killing the TIF/LRF mutant kills the TIF/LOF mutant. The exception is that a term with equivalent LIFs is processed if and only if for each equivalent LIF, the literal being inserted is from a single-literal term. Consider $ab + cd + e$ and note $ab\sim e + cd + e$ is equivalent. However, an LRF where $\sim e$ replaces a is an LOF for a and is treated as such.

Table 29 shows in tabular format an example of a TRF/LIF, TIF/LRF, and TIF/LOF being seeded into a minimal DNF predicate for the purpose of detecting other faults.

Table 29 TRF-TIF Faults [18]

Fault	Description
Term Reference Fault / Literal Insertion Fault (TRF/LIF)	Replacing a term with one or more terms to guarantee detecting LIFs: $ab + cd$ implemented as $abc + ab\sim d + cd$ to detect the LIFs $abc + cd$ and $ab\sim d + cd$.
Term Insertion Fault / Literal Reference Fault (TIF/LRF)	Inserting a term containing all literals to guarantee detecting LRFs: $ab + ac + ad$ implemented as $ab + ac + ad + a\sim b\sim c\sim d$ to detect the LRFs $a\sim c + ac + ad$, $a\sim d + ac + ad$, $ab + a\sim b + ad$, $ab + a\sim d + ad$, $ab + ac + a\sim b$, $ab + ac + a\sim c$.
Term Insertion Fault / Literal Omission Fault (TIF/LOF)	Inserting a term containing all literals to guarantee detecting LOFs: $ab + cd$ implemented as $ab + cd + \sim ab\sim cd$ to detect the LOFs $b + cd$ and $ab + d$.

For any given literal in a minimal DNF predicate, it is never necessary to generate both a TIF/LRF mutant or a TIF/LOF mutant to guarantee detection of all faults in Lau and Yu's fault hierarchy (assuming all non-equivalent mutants are killed) [18]. This further reduces the number of mutants that need to be generated.

When no equivalent LIFs exist for a term, any test detecting an LIF where literal y is inserted into a term X will also detect an LRF where literal y replaces any literal x in term X . The reason is that when no equivalent LIFs exist for term X , every LRF in term X can be detected by a UTP that is needed to detect an LIF in term X . Thus, TIF/LRF mutants are not needed for any literals in term X because any test set that kills all TRF/LIF mutants for term X is guaranteed to kill all LRFs for literals in term X [18].

When an equivalent LIF does exist by inserting a literal y into term X , any test detecting an LRF where literal y replaces a literal x in term X also detects an LOF for literal x . The reason why is that when an equivalent LIF exists for term X , there is an LRF for each literal x in term X that can only be detected by a specific NFP for literal x and any NFP for literal x is guaranteed to detect an LOF for literal x [18]. Thus, TIF/LOF mutants are not needed for any literals in term X because any test set that kills all TIF/LRF mutants for literals in term X is guaranteed to kill all LOFs for literals in term X .

The algorithm used to produce TRF-TIF logic mutations is given next.

TRF-TIF Logic Mutation Algorithm [18]

Input is a minimal DNF predicate; Output is a mutant set M

```
if predicate is a single term
  add TIF/LOF mutants and one FALSE mutant to M;
else if every term is a single literal
  add TRF/LIF mutants and one TRUE mutant to M;
else
  for each term
    if the term contains all literals
      add a TOF mutant to M;
    else
      add TRF/LIF mutants to M;
  end for
  if (Number of TIF/LOF mutants + Number of TIF/LRF mutants < Number of false
    points)
    for each literal in a term with no equivalent LIF mutants or in a term where all
      equivalent LIF mutants involve inserting literals from single-literal terms
      if the TIF/LOF mutant is not killed by a test killing a mutant in M
        add it to M;
    end for
    for each literal in each term with an equivalent LIF mutant not formed by inserting a
      literal from a single literal-term
      if the corresponding LRF mutant is not equivalent
        if the corresponding TIF/LRF mutant is not killed by a test killing a mutant in
          M
          add it to M;
        else if the corresponding TIF/LOF mutant is not killed by a test killing a mutant in
          M
          add it to M;
      end for
    else
      add all TIF mutants to M;
```

Extended Fault Hierarchy

Figure 9 supplements Lau and Yu's fault hierarchy with the faults in Table 29 and faults produced by typical logic mutation operators. Fault detection relationships for the

faults in Lau and Yu's hierarchy are proved by Lau and Yu [30]. Kaminski and Ammann [18] give proofs of the fault detection relationships for the other faults in Figure 9.

Legend for Figure 9

Solid-lined boxes exist in Lau and Yu's fault hierarchy.

Dashed-lined boxes do not exist in Lau and Yu's fault hierarchy.

Thin-lined boxes represent faults that have corresponding mutation operators in a typical logic mutation approach.

Thick-lined boxes represent faults that do not have corresponding mutation operators in a typical logic mutation approach.

Solid arrows represent guaranteed fault detection.

Dashed arrows represent fault detection that holds if and only if the source fault does NOT result in an equivalent mutant and (if the source fault itself is a destination fault in a dashed arrow connection) the source's source fault DOES result in an equivalent mutant.

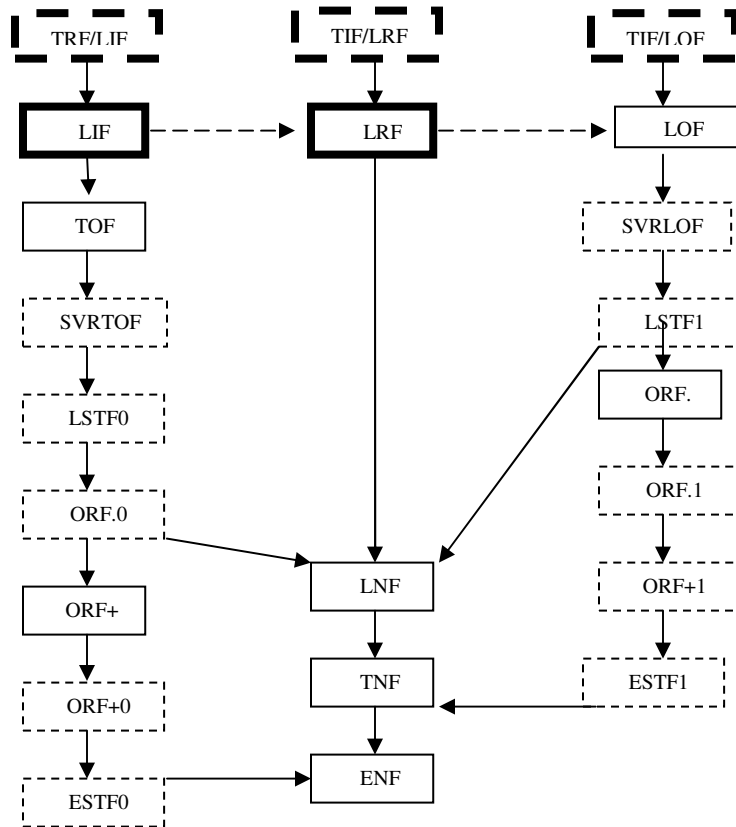


Figure 9 Extended Fault Hierarchy [18]

Figure 9 indicates that (1) generating an LRF mutant is unnecessary if the corresponding LIF mutant is non-equivalent and (2) generating an LOF mutant is unnecessary if the corresponding LIF mutant is equivalent but the corresponding LRF mutant is not. When a non-equivalent LIF mutant exists, an input killing it will kill a corresponding LRF mutant. However, if an equivalent LIF mutant exists, killing all non-equivalent LIF mutants does not guarantee killing all LRF mutants. In the absence of an equivalent LIF mutant, only TRF/LIF and TIF/LOF mutants are needed. When an equivalent LIF mutant occurs, it is only necessary to produce TIF/LRF mutants for the corresponding LRFs.

If an equivalent LIF mutant occurs, an input killing a corresponding LRF mutant will kill a corresponding LOF mutant. The proof is as follows. First it must be established that if an equivalent LIF occurs for term X , it is infeasible to let some literal y in some term Y (but not in X) obtain both 0 and 1 in a UTP for X . Inserting a literal into a term can make a TRUE term FALSE but cannot make a FALSE term TRUE. Thus, when an LIF mutant is equivalent, there is no UTP for X that makes Xy FALSE in the faulty predicate because making Xy FALSE in the faulty predicate makes term Y TRUE. Thus, y cannot be assigned both truth values in a UTP for X as Y would be TRUE for one of the truth value assignments to y , establishing the infeasibility. If it is infeasible to let an external literal obtain both 0 and 1 in a UTP for a term, an LRF mutant exists that can only be killed by a corresponding NFP. Since an LOF mutant can be killed by any NFP for the omitted literal, the corresponding NFP killing an LRF mutant kills a corresponding LOF mutant. It has been shown that (1) when an equivalent LIF mutant

occurs, the infeasibility condition holds and (2) when this condition holds, a test killing an LRF mutant kills a corresponding LOF mutant. This completes the proof.

It has been established that (1) when no equivalent LIF mutants occur, no TIF/LRF mutants are needed and (2) when an equivalent LIF mutant occurs and no equivalent corresponding LRF mutants occur, a corresponding LOF mutant is not needed. Thus, for any literal it is never necessary to generate both TIF/LRF and TIF/LOF mutants.

The extended fault hierarchy implies that if no LIFs or LRFs exist (each literal is in each term) an LOF covers the faults under an LRF but the TOF is needed to cover the faults under an LIF. The TRF-TIF approach generates TOF mutants only in this case. A TOF mutant for a particular term is guaranteed to be killed when a non-equivalent LIF mutant for that term is killed as there is a many-to-one relationship between an LIF and TOF. Many LIFs correspond to one TOF such that if an equivalent LIF corresponding to a TOF occurs, there is also a corresponding non-equivalent LIF. Consider $ab + b\sim c + \sim bc$ and the predicates $abc + b\sim c + \sim bc$ and $ab\sim c + b\sim c + \sim bc$. The first fault is equivalent, but the second is not. Both LIFs correspond to the same TOF for ab . The relationship between an LIF and LRF is a one-to-many relationship. The equivalent LIF yielding $abc + b\sim c + \sim bc$ corresponds to two different LRFs: $cb + b\sim c + \sim bc$ and $ac + b\sim c + \sim bc$. No non-equivalent LIF exists corresponding to either LRF. The one-to-many relation between the LIF and LRF is the reason for the dashed arrow between the LIF and LRF in Figure 9.

Mutation Efficiency

Two measurements of mutation efficiency are introduced by Kaminski and Ammann [18] to compare mutation approaches: *Mutation Efficiency Difference* (MED) and *Mutation Efficiency Ratio* (MER). The terms “a mutant detects a fault” and “faults detected” are used to mean that either (1) a non-equivalent mutant forces the generation of data that detects the fault or (2) an equivalent mutant is detected. MED and MER are defined as follows:

$$\text{MED} = \text{Faults Detected} - \text{Mutants Generated}$$

$$\text{MER} = \text{Faults Detected} / \text{Mutants Generated}$$

The faults below the TRF/LIF, TIF/LRF and TIF/LOF in the extended fault hierarchy are listed in Table 30 for $ab + b\sim c + \sim bc$. A typical logic mutation approach produce 61 mutants and detects 60 faults so MED = -1 and MER = 0.98. The reason why 60 faults are detected is that a typical logic mutation approach will produce one equivalent mutant, namely an LRF where literal c replaces literal b in term ab to yield $ac + b\sim c + \sim bc$. A typical logic mutation approach produces the same fault multiple times, but each fault is considered different to establish a baseline of MED = 0 and MER = 1 (when no equivalent mutants are produced), which simplifies the analysis without bias.

The TRF-TIF approach detects 73 faults (the original 61 plus six LIFs and six LRFs where a literal is replaced by the negation of another literal). The TRF-TIF approach produces seven mutants so MED = 73 - 7 = 66 and MER = 73 / 7 = 10.43. TRF/LIFs are $ab\sim c + b\sim c + \sim bc$, $ab + b\sim ca + \sim bc$, $ab + b\sim c + \sim bca$ and $ab + b\sim c + \sim bc\sim a$. Note that $abc + b\sim c + \sim bc$ and $ab + b\sim c\sim a + \sim bc$ are equivalent to $ab + b\sim c +$

$\sim bc$ and are not produced. The TRF-TIF algorithm has an optimization to create all TIF mutants (inserting a term not in the predicate) if the number of FALSE points are less than or equal to the number of TIF/LRF and TIF/LOF mutants that would otherwise be generated. For this example, the sum of TIF/LRF and TIF/LOF mutants is six, but only three FALSE points exist. These points (000, 100, 011) result in the insertion of three terms individually ($\sim a \sim b \sim c$, $a \sim b \sim c$, $\sim abc$) to form three TIF mutants. Only eight inputs exist for the predicate, so the TRF-TIF approach yields little savings compared to exhaustive testing. The TRF-TIF approach guarantees the number of mutants never exceeds 2^n where n is the number of unique literals. The number of UTPs (and hence the number of TRF/LIF mutants) is always less than or equal to the number of TRUE points and the number of NFPs (and hence the number of TIF/LRF and TIF/LOF mutants) is always less than or equal to the number of FALSE points.

Table 30 Faults for $ab + b\sim c + \sim bc$

Fault	Example	Total
LRF	$ac + b\sim c + \sim bc$	12
LIF	$abc + b\sim c + \sim bc$	6
LOF	$b + b\sim c + \sim bc$	6
LSTF0	$0b + b\sim c + \sim bc$	6
LSTF1	$1b + b\sim c + \sim bc$	6
LNF	$\sim ab + b\sim c + \sim bc$	6
TOF*	$b\sim c + \sim bc$	4
SVRTOF	$ab + c\sim c + \sim bc$	4
ORF.	$a + b + b\sim c + \sim bc$	3
ORF.0	$0 + b\sim c + \sim bc$	3
ORF.1	$1 + b\sim c + \sim bc$	3
TNF	$\sim(ab) + b\sim c + \sim bc$	3
SVRLOF	$bb + b\sim c + \sim bc$	2
ORF+	$abb\sim c + \sim bc$	2

Fault	Example	Total
ORF+0	$0 + \sim bc$	2
ORF+1	$1 + \sim bc$	2
ESTF0	0	1
ESTF1	1	1
ENF	$\sim(ab + b\sim c + \sim bc)$	1

* 4 TOFs exist even though only 3 terms exist because a typical logic mutation approach performs a TOF for term $b\sim c$ twice since term $b\sim c$ has an OR operator on each side of it

7.2 Mutant Set Size

An empirical study was undertaken by Kaminski and Ammann [18] to evaluate TRF-TIF mutant set size for a program containing the 19 predicates in Appendix B. The TRF-TIF tool generates mutants and the necessary assignments to literals in the predicates. The predicates had from 5 to 13 unique literals and were converted manually to minimal DNF. Predicate number 12 is excluded because it was missing a right parenthesis detected by Weyuker et al. [46]. For each predicate, mutant set size for TRF-TIF logic mutation was determined. An example of TRF-TIF logic mutations for one of the predicates in the study is given in Appendix I. Table 31 below shows the results. Note that the data in Table 31 is the same as that in Table 10. The reason for this is to weakly kill all TRF-TIF logic mutants, it is necessary (but no sufficient) to satisfy the Minimal-MUMCUT criterion. That is, each Minimal-MUMCUT test requirement maps to a TRF-TIF logic mutant.

Table 31 Number of TRF-TIF Logic Mutants Generated [18]

Predicate	TRF-TIF	2^n	Percentage
1	27	128	21.09%
2	81	512	15.82%
3	148	4096	3.61%
4	9	32	28.13%
5	34	512	6.64%
6	62	2048	3.03%
7	62	1024	6.05%
8	36	256	14.06%
9	16	128	12.50%
10	62	8192	0.76%
11	61	8192	0.74%
12	N/A	N/A	N/A
13	17	4096	0.42%
14	22	128	17.19%
15	39	512	7.62%
16	104	4096	2.54%
17	39	2048	1.90%
18	48	1024	4.69%
19	16	256	6.25%
20	14	128	10.94%
Sum	897	37408	
Average	47.21	1968.84	2.40%

7.3 Equivalent Mutant Set Size

Equivalent mutants are a problem because no test input can kill them. In order to strongly kill a mutant, the mutated statement must be reached (reachability), the program state of the mutant must differ from that of the original program after execution of the mutated statement (infection) and the difference in program state must propagate to the output (propagation) [2]. A mutant is weakly equivalent if infection can never occur whereas a mutant is strongly equivalent if propagation can never occur. Thus, all weakly

equivalent mutants are strongly equivalent, but not all strongly equivalent mutants are weakly equivalent as it is possible that infection occurs but not propagation.

When a test set does not strongly kill a mutant, the tester does not know if more tests should be added to strongly kill it (if it is not strongly equivalent) or if the tester should discard the mutant (if it is strongly equivalent). The TRF-TIF logic mutation tool will never generate a weakly equivalent logic mutant when the Boolean space is complete (assuming reachability holds). Thus, the mutation score (percentage of mutants strongly killed) will be a more accurate measure of the quality of the test data because the score will not be affected by the presence of weakly equivalent mutants.

When the Boolean space is incomplete (some of the points are infeasible), weakly equivalent logic mutants can be generated by the TRF-TIF mutation tool, even when reachability holds. To address this problem, the tool permits users to mark predicates in the source code where infeasibilities arise for combinations of literal values. For example, consider a predicate that determines if the length of any side of a triangle is greater than or equal to the sum of the lengths of the other two sides. Let s_1 be one side, s_2 be a second side and s_3 be a third side. Let literal x be $s_1 \geq s_2 + s_3$, literal y be $s_2 \geq s_1 + s_3$ and literal z be $s_3 \geq s_1 + s_2$. The predicate is thus $x \text{ OR } y \text{ OR } z$. Only one of x , y or z can be TRUE. Thus infeasibilities arise and the complete Boolean space is not achievable. When using the TRF-TIF tool the tester can specify the infeasible combinations in a comment in the source code prior to the actual predicate. This prevents weakly equivalent logic mutants from being generated (assuming reachability).

When the TRF-TIF tool is informed that certain points in the Boolean space are infeasible, the tool will replace any weakly equivalent mutant that would otherwise be generated with a weakly non-equivalent mutant (assuming reachability) so as to still guarantee fault detection of all faults in Lau and Yu's fault hierarchy. For example, consider predicate $ab + cd$. TRF/LIF mutations for term ab include $p' = abc + ab\sim d + cd$ and $p'' = ab\sim c + abd + cd$. These mutants are weakly non-equivalent (assuming reachability). However if it is infeasible for literals a , b and c to all be TRUE, then the p'' mutant is weakly equivalent because the only point that can kill the p'' mutant is 1110. If the infeasible combination is specified in the source code, the TRF-TIF tool will recognize that such a mutant is weakly non-equivalent.

When the TRF-TIF tool automatically detects a weakly equivalent mutant due to an infeasibility in the Boolean space, the mutant cannot be simply discarded. In the example above, discarding the p'' mutant causes the LIF where literal d is inserted into term ab to be missed. The reason is that the p'' mutant (if killed) guarantees detecting both the LIF where literal $\sim c$ is inserted into term ab and the LIF where literal d is inserted into term ab . If literals a , b and c cannot be all TRUE, the LIF where literal $\sim c$ is inserted into term ab is weakly equivalent but the LIF where literal d is inserted into term ab is not. Thus, the TRF-TIF tool will change the p'' mutant to be $p'' = abc + abd + cd$. This mutant has the dual property of being both weakly non-equivalent (assuming reachability) and guaranteeing detection of the LIF where literal d is inserted into term ab (assuming it is killed).

7.4 Single Minimal DNF Fault Detection

For software containing minimal DNF predicates, tests that weakly kill all TRF-TIF mutants are guaranteed to detect all 9 single fault types in Lau and Yu's fault hierarchy. This is because to kill the mutants produced by the TRF-TIF algorithm it is necessary to satisfy the Minimal-MUMCUT criterion. This fault detection can also be proven by the fault detection relations between the TRF/LIF and the LIF, the TIF/LRF and the LRF and the TIF/LOF and the LOF. The reason is that detecting the LIF, LRF and LOF guarantees detection of all the other faults in the hierarchy.

7.5 Double Minimal DNF Fault Detection

For software containing minimal DNF predicates, tests that weakly kill all TRF-TIF mutants are guaranteed to detect 84 of the 92 double fault types in Lau and Yu's fault hierarchy. Again, this is because to kill the mutants produced by the TRF-TIF algorithm it is necessary to satisfy the Minimal-MUMCUT criterion. This fault detection can also be proven by the fault detection relations between the TRF/LIF and the LIF and the TIF/LOF and the LOF. The reason is that an LIF can only be detected by a UTP and an LOF can only be detected by an NFP, and any test set that includes a UTP for each term and an NFP for each literal guarantees detection of 84 of the 92 double fault types in Lau and Yu's fault hierarchy.

7.6 General Fault Detection

The ability of a test set that weakly kills all TRF-TIF logic mutants to strongly kill general mutants (both logic and non-logic mutants) is evaluated in an empirical study described in sections 9.5 and 10.5. Discussion of this study is delayed until these sections

since muJava (discussed in Chapter 9) and SQLMutation (discussed in Chapter 10) were used to seed general faults.

8 Comparison of TRF-TIF Logic Mutation with Typical Logic Mutation

This chapter is the first of three comparing the TRF-TIF logic mutation approach with other mutation tools/approaches. The focus of this chapter is comparing the TRF-TIF logic mutation approach with a typical logic mutation approach, which corresponds to the third row in Table 5. A typical logic mutation approach uses all mutation operators in the text by Ammann and Offutt [2] that can be applied to a minimal DNF predicate containing at least two literals. These mutation operators include the Scalar Variable Replacement (SVR) operator, the Unary Operator Insertion (UOI) operator, the Unary Operator Deletion (UOD) operator, and the Conditional Operator Replacement (COR) operator [2]. In addition, the Expression Stuck at Fault (ESTF) and Literal Stuck at Fault (LSTF) operators are included when referring to typical logic mutation, as these are also common logic mutation operators. Each section in this chapter corresponds to a cell in Table 5. Section 8.1 corresponds to cell 3a (mutant set size comparison), section 8.2 corresponds to cell 3b (equivalent mutant set size comparison), section 8.3 corresponds to cell 3c (single minimal DNF fault detection comparison), and section 8.4 corresponds to cell 3d (double minimal DNF fault detection comparison). The overriding theme of this chapter is that TRF-TIF logic mutation provides both better minimal DNF fault detection and reduced mutant set size when compared to a typical logic mutation approach.

8.1 Mutant Set Size (Contribution 3a)

An empirical study was undertaken by Kaminski and Ammann [18] to compare TRF-TIF logic mutation with typical logic mutation. This study used the same program and predicates described in section 7.2. For each predicate, the mutant set size for TRF-TIF logic mutation and typical logic mutation was determined. The study found that TRF-TIF mutant set size was about 6% of the logic mutant set size generated by a typical logic mutation approach (**Contribution 3a**). Table 32 shows the results.

Table 32 Number of Typical Logic Mutants Generated [18]

Predicate	TRF-TIF	Typical	Percentage	2^n
1	27	409	6.60%	128
2	81	1694	4.78%	512
3	148	3297	4.49%	4096
4	9	84	10.71%	32
5	34	454	7.49%	512
6	62	1048	5.92%	2048
7	62	1026	6.04%	1024
8	36	482	7.47%	256
9	16	196	8.16%	128
10	62	1204	5.15%	8192
11	61	1267	4.81%	8192
12	N/A	N/A	N/A	N/A
13	17	268	6.34%	4096
14	22	228	9.65%	128
15	39	521	7.49%	512
16	104	1674	6.21%	4096
17	39	580	6.72%	2048
18	48	652	7.36%	1024
19	16	302	5.30%	256
20	14	171	8.19%	128
Sum	897	15557		
Average	47.21	818.79	5.77%	

These results can be explained by a number of factors. One factor is that typical logic mutation does not take advantage of Lau and Yu's fault hierarchy. In other words, typical logic mutation generates mutants that correspond to faults low in the hierarchy. These mutants are guaranteed to be killed by tests that kill other generated mutants that correspond to faults higher in the hierarchy. Thus excess mutants are generated.

Another factor is that typical logic mutation can generate the same semantic logic mutant multiple times. As an example of the same semantic logic mutant being generated multiple times can be seen by considering the typical Scalar Variable Replacement (SVR) mutation operator. This operator replaces each variable reference with every other variable reference of the same type. The SVR mutation operator produces an LOF or a TOF when replacing a literal with another literal in the term. In $a \sim b \sim c \sim d + e$, replacing a with b , c , or d results in a TOF for $a \sim b \sim c \sim d$. In $abcd + e$, replacing a with b , c , or d results in an LOF for a . So one problem with the SVR mutation operator is that it produces the same TOF and LOF mutants multiple times. The Conditional Operator Replacement (COR) mutation operator can also produce the same mutant twice as one mutation it generates is to replace each occurrence of "operand AND operand" with each operand. In $abc + d$ the AND between a and b yields $f' = ac + d$ and $f'' = bc + d$. The AND between b and c yields $f''' = ab + d$ and $f'''' = ac + d$. f' and f'''' are identical. For a term with n literals (for $n > 2$), the COR mutation operator produces $n - 2$ identical LOF mutants as each literal except the first and last is processed twice. Duplicate processing also occurs for terms, resulting in $m - 2$ identical TOF mutants for a predicate with m terms (for $m > 2$).

A third factor that increases typical logic mutant set size is the number of equivalent mutants typical logic mutation generates, discussed next.

8.2 Equivalent Mutant Set Size (Contribution 3b)

This section presents a theoretical contribution by proving that a TRF-TIF mutant set contains the same or fewer weakly equivalent than a typical logic mutants set.

Theorem 5 (Contribution 3b): TRF-TIF vs. Typical Logic Mutation Equivalent Mutant Set Size

For software with minimal DNF predicates, a TRF-TIF mutant set contains the same number as or fewer weakly equivalent mutants than a typical logic mutant set (assuming any infeasible combinations of values of unique literals are specified).

Proof:

If reachability cannot be achieved for a given mutant, then it will be equivalent no matter if it is a TRF-TIF mutant or a typical logic mutant. If a mutant is reachable, the TRF-TIF mutation approach guarantees it will not be weakly equivalent (assuming any infeasible combinations of values of unique literals are specified) because TRF-TIF mutation uses an exclusive-OR algorithm to automatically detect equivalent mutants. Thus, if a tool that produces typical logic mutants can produce at least one weakly equivalent mutant, then it is established that a TRF-TIF mutant set contains the same number as or fewer weakly equivalent mutants than a typical logic mutant set (assuming any infeasible combinations of values of unique literals are specified). The SVR typical logic mutation operator can produce an equivalent mutant as follows. Consider the predicate $ab + b!c + !bc$. This

operator will mutate this predicate to $ac + b!c + !bc$ by replacing literal b in term ab with literal c . The original predicate and the mutated predicate are semantically equivalent.

End Proof

8.3 Single Minimal DNF Fault Detection (Contribution 3c Parts I and II)

This section presents a theoretical contribution by proving the single Minimal DNF fault detection capability of a typical logic mutation test set.

Theorem 6 (Contribution 3c Part I): TRF-TIF vs. Typical Logic Mutation Single Minimal DNF Fault Detection

For software containing minimal DNF predicates, tests that weakly kill all typical logic mutants are guaranteed to detect 7 of the 9 single fault types in Lau and Yu's fault hierarchy (all but the LRF and the LIF).

Proof:

Typical logic mutation does not include mutation operators that cover the LRF and LIF [18]. Thus, tests that weakly kill all typical logic mutants are not guaranteed to detect the LRF and the LIF. However, the COR typical logic mutation operator produces TOFs and LOFs. A test set detecting TOFs and LOFs will detect all faults in the fault hierarchy except for LIFs and LRFs so 7 of the 9 single fault types are guaranteed to be detected.

End Proof

An LRF is partially covered by the Scalar Variable Replacement (SVR) mutation operator which replaces each variable reference by every other variable reference of the same type. For example, in $ab + cd$, the SVR mutation operator replaces literal a with literal c , but it would not replace literal a with literal $\sim c$. Typical logic mutation does

include mutations that correspond to a TOF (which can only be detected by a UTP) and an LOF (which can only be detected by an NFP) [18].

The Conditional Operator Replacement (COR) typical mutation operator replaces each occurrence of “operand AND operand” with each operand. For example, in $abc + d$ the AND between a and b yields $f' = ac + d$ and $f'' = bc + d$. The COR mutation operator covers the ENF, ORF+, ORF., LOF and TOF. The COR mutation operator produces additional ORFs not in Lau and Yu’s fault hierarchy by replacing the OR and AND operators with 0 and 1. These are also known as Stuck at Faults because terms or literals are stuck at 0 or 1.

Other typical logic mutation operators are the Unary Operator Insertion (UOI) operator and Unary Operation Deletion (UOD) operator (which insert and delete negations, respectively). These operators covers the ENF, TNF and LNF [18]. The Literal Constant Replacement mutation operator produces Literal Stuck at Faults, replacing each literal with 0 and 1 (LSTF0 and LSTF1). To achieve predicate coverage a mutation operator is commonly used to produce Stuck at Faults for the entire predicate. The corresponding faults for these operators are in Table 33.

All faults in Table 33 are covered by faults in Lau and Yu’s fault hierarchy in that a test set that detects all of the faults in Lau and Yu’s fault hierarchy will also detect all faults in Table 33. The reason is as follows. The LSTF0 and ORF.0 are equivalent to a TOF and LSTF1 is equivalent to an LOF. ORF.1 is equivalent to performing two LOFs and ORF+0 is equivalent to performing two TOFs. Since any test that detects an LOF for a literal detects a double LOF involving that literal, any test that detects an LOF also

detects a corresponding ORF.1. Since any test that detects a TOF for a term detects a double TOF involving that term, any test that detects a TOF also detects a corresponding ORF+0. ORF+1 is equivalent to ESTF1, any FALSE point detects ESTF1 and any TRUE point detects ESTF0. The point is that the mutants produced by all of the typical logic mutation operators are not necessary if mutants are generated that when weakly killed, guarantee TOF and LOF detection. Figure 9 shows that tests that weakly kill all TRF-TIF mutants will also weakly kill all TOF and LOF mutants.

Table 33 Faults Produced by Typical Mutation Operators that are not in Lau and Yu's Fault Hierarchy [18]

Fault	Description
Expression Stuck at Fault 0 (ESTF0)	Predicate implemented as 0: $a + b$ implemented as 0.
Expression Stuck at Fault 1 (ESTF1)	Predicate implemented as 1: $a + b$ implemented as 1.
Literal Stuck at Fault 0 (LSTF0)	Literal implemented as 0: $ab + c$ implemented as $a0 + c$.
Literal Stuck at Fault 1 (LSTF1)	Literal implemented as 1: $ab + c$ implemented as $a1 + c$.
Operator Reference Fault +0 (ORF+0)	Replacing OR with 0: $a + b + c$ implemented as $0 + c$ (generated by COR operator).
Operator Reference Fault +1 (ORF+1)	Replacing OR with 1: $a + b + c$ implemented as $1 + c$ (generated by COR operator).
Operator Reference Fault .0 (ORF.0)	Replacing AND with 0: $abc + d$ implemented as $0c + d$ (generated by COR operator).
Operator Reference Fault .1 (ORF.1)	Replacing AND with 1: $abc + d$ implemented as $1c + d$ (generated by COR operator).
Scalar Variable Replacement TOF	Replacing a literal with another literal in the term such that a TOF occurs: $a \sim b$ implemented as $b \sim b$.
Scalar Variable Replacement LOF	Replacing a literal with another literal in the term such that an LOF occurs: ab implemented as aa .

In the empirical study described in section 8.1, fault detection was compared in all predicates for all the faults below the TRF/LIF, TIF/LRF and TIF/LOF in the extended fault hierarchy. It was found that tests that weakly kill all typical logic mutants guaranteed detection of about 76% of the single logic faults (as opposed to 100% for tests that weakly kill all TRF-TIF mutants) (**Contribution 3c Part II**). Table 34 shows the results.

Table 34 Number of Faults Detected by Typical Mutation [18]

Predicate	Typical	TRF-TIF	Percentage	2 ⁿ
1	409	455	89.89%	128
2	1694	1814	93.38%	512
3	3297	4487	73.48%	4096
4	84	116	72.41%	32
5	454	714	63.59%	512
6	1048	1136	92.25%	2048
7	1026	1214	84.51%	1024
8	482	482	100.00%	256
9	196	196	100.00%	128
10	1204	1420	84.79%	8192
11	1267	1747	72.52%	8192
12	N/A	N/A	N/A	N/A
13	268	506	52.96%	4096
14	228	348	65.52%	128
15	521	849	61.37%	512
16	1674	2747	60.94%	4096
17	580	824	70.39%	2048
18	652	934	69.81%	1024
19	302	386	78.24%	256
20	171	187	91.44%	128
Sum	15557	20562		
Average	818.79	1082.21	75.66%	

Table 35 shows the overall MED and MER. On average, the TRF-TIF logic mutation approach detected 1,035 more faults than mutants generated. The ratio of faults detected to mutants generated was nearly 23 times higher for the TRF-TIF logic mutation approach than for a typical logic mutation approach.

Table 35 MED and MER [18]

	Typical	TRF-TIF
MED	0	1035
MER	1	22.92

Table 36 displays TRF-TIF MED and MER results by predicate. Note from this table that the number of unique literals had a large impact on MED (the MED increased as the number of unique literals increased) but minimal impact on MER

Table 36 TRF-TIF MED and MER [18]

Predicate	MED	MER	2 ⁿ
1	428	16.85	128
2	1733	22.40	512
3	4339	30.32	4096
4	107	12.89	32
5	680	21.00	512
6	1074	18.32	2048
7	1152	19.58	1024
8	446	13.39	256
9	180	12.25	128
10	1358	22.90	8192
11	1686	28.64	8192
12	N/A	N/A	N/A

Predicate	MED	MER	2^n
13	488	29.76	4096
14	326	15.82	128
15	810	21.77	512
16	2643	26.41	4096
17	785	21.13	2048
18	886	19.46	1024
19	370	24.13	256
20	173	13.36	128
Sum	19665		
Average	1035	22.92	

8.4 Double Minimal DNF Fault Detection (Contribution 3d Parts I and II)

This section presents a theoretical contribution by proving the double Minimal DNF fault detection capability of a typical logic mutation test set.

Theorem 7 (Contribution 3d Part I): TRF-TIF vs. Typical Logic Mutation Double Minimal DNF Fault Detection

For software containing minimal DNF predicates, tests that weakly kill all typical logic mutants are guaranteed to detect 84 of the 92 double fault types in Lau and Yu's fault hierarchy.

Proof:

In order to kill the mutants produced by the COR typical mutation operator, it is necessary to have a UTP for each term and an NFP for each literal because the COR typical mutation operator produces TOFs and LOFs [18]. Lau et al. [27, 28, 29] show that

any strategy that includes at least one UTP for each term and one NFP for each literal is guaranteed to detect 84 of the 92 double fault types in Lau and Yu's fault hierarchy.

End Proof

In the empirical study described in section 8.1, double fault detection was compared. Since the PCUTPNFP criterion is feasible for all predicates in the study and since TRF-TIF logic mutation subsumes the Minimal-MUMCUT criterion, for these predicates tests weakly killing all TRF-TIF mutants guaranteed detection of 91 of the 92 double fault types in Lau and Yu's fault hierarchy. However, for these predicates tests that weakly kill all typical logic mutants still detected just 84 of the 92 double fault types as a test set weakly killing all typical logic mutants missed detecting faults corresponding to the 8 double fault types not guaranteed to be detected (**Contribution 3d Part II**).

9 Comparison of TRF-TIF Logic Mutation with muJava

This chapter is the second of three comparing the TRF-TIF logic mutation approach with other mutation tools/approaches. The focus of this chapter is comparing the TRF-TIF logic mutation approach with muJava, which corresponds to the fourth row in Table 5. muJava is a tool for mutating Java programs. When referring to muJava mutants for comparison of fault detection and mutant set size, the author refers to all of the mutants generated by muJava and not solely the logic mutants. Each section in this chapter corresponds to a cell in Table 5. Section 9.1 corresponds to cell 4a (mutant set size comparison), section 9.2 corresponds to cell 4b (equivalent mutant set size comparison), section 9.3 corresponds to cell 4c (single minimal DNF fault detection comparison), and section 9.4 corresponds to cell 4d (double minimal DNF fault detection comparison), and section 9.5 corresponds to cell 4e (general fault detection). The overriding theme of this chapter is that TRF-TIF logic mutation provides better minimal DNF fault detection and reduced mutant set size when compared to muJava, and that TRF-TIF logic mutation is effective at producing mutants that when killed, also kill a high percentage of non-logic mutants. The rest of this section describes an empirical study from which contributions in subsequent sections are derived.

An empirical evaluation [23] was conducted to ascertain if logic mutation testing can be used to reduce the costs of general mutation testing while maintaining most of its benefits. Specifically, the study was performed to answer these questions:

- 1) Does TRF-TIF logic mutation produce fewer mutants than muJava with minimal impact on general fault detection?
- 2) Does TRF-TIF logic mutation produce fewer mutants than muJava while improving logic fault detection?
- 3) Does TRF-TIF logic mutation produce fewer equivalent mutants than muJava?
- 4) Can a metric be established concerning a property of the source code under test that will predict when killing all TRF-TIF mutants is likely to kill general mutants?

Thirty small Java programs (average of 15 LOC per program) were selected for the study. Four of the programs appear in a textbook on software testing by Ammann and Offutt [2]. The code for these programs is in Appendix J. The author supplemented these programs with 26 additional programs, each of which corresponds to a static utility method in the Arrays or Collections Java classes (J2SE 1.7). These programs are necessarily small as this makes it feasible to manually determine the equivalent mutants generated by muJava. The code for the Collections and Arrays classes is accessible at:

<http://www.docjar.com/html/api/java/util/Collections.java.html>

and

<http://www.docjar.com/html/api/java/util/Arrays.java.html>

For the Collections class, a sample of deterministic static utility methods that contained at least one “if” statement were included in the study. Most of these methods

focused on sorting and searching. For the Arrays class, a similar set of static utility methods was selected except that sorting and searching methods were excluded to avoid redundancy since such methods were similar to the Collections methods. Also, if multiple methods in the Arrays class existed that differed only in the type of primitive array(s) taken as argument(s), only one such method was selected. For example, the Arrays class has a different equals static utility method for each of the following array types: *long[]*, *int[]*, *char[]*, *short[]*, *byte[]*, *boolean[]*, *double[]*, *float[]* and *Object[]*. In this case the method that uses *Object[]* was chosen as well as the method using *long[]*. Table 37 gives the class name and method name for each of the 26 Collections and Arrays programs.

Table 37 Arrays and Collections Programs [23]

Class	Method Name
Arrays	rangeCheck(int arrayLen, int fromIndex, int toIndex)
Arrays	equals(long[] a, long[] a2)
Arrays	equals(Object[] a, Object[] a2)
Arrays	deepEquals(Object[] a1, Object[] a2)
Arrays	toString(long[] a)
Arrays	toString(Object[] a)
Arrays	deepToString(Object[] a)
Arrays	deepToString(Object[] a, StringBuilder buf, Set<Object[]> dejaVu)
Collections	binarySearch(List<? extends Comparable<? super T>> list, T key)
Collections	indexedBinarySearch(List<? extends Comparable<? super T>> list, T key)
Collections	iteratorBinarySearch(List<? extends Comparable<? super T>> list, T key)
Collections	get(ListIterator<? extends T> i, int index)
Collections	binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
Collections	indexedBinarySearch(List<? extends T> l, T key, Comparator<? super T> c)
Collections	iteratorBinarySearch(List<? extends T> l, T key, Comparator<? super T> c)
Collections	reverse(List<?> list)
Collections	fill(List<? super T> list, T obj)
Collections	copy(List<? super T> dest, List<? extends T> src)
Collections	min(Collection<? extends T> coll)
Collections	min(Collection<? extends T> coll, Comparator<? super T> comp)

Class	Method Name
Collections	max(Collection<? extends T> coll)
Collections	max(Collection<? extends T> coll, Comparator<? super T> comp)
Collections	rotate(List<?> list, int distance)
Collections	rotate1(List<T> list, int distance)
Collections	rotate2(List<?> list, int distance)
Collections	replaceAll(List<T> list, T oldVal, T newVal)

9.1 Mutant Set Size (Contribution 4a Parts I and II)

The study used the TRF-TIF tool to generate logic mutants and muJava to generate general mutants. To reduce bias, the test data to weakly kill all TRF-TIF mutants was generated prior to the generation of muJava mutants. The number of mutants generated by each tool was determined and the results are displayed in Table 36. In this table, the names of Arrays and Collections programs in Table 37 are referred to by the letter A if the program is an Arrays method and by the letter C if the program is a Collections method. The number after the letter corresponds to the order the method appears in Table 37. So for example, A1 refers to the Arrays rangeCheck method and C1 refers to the Collections binarySearch method that takes two arguments.

Table 38 Number of Software Mutants [23]

Program	Number of TRF-TIF Logic Mutants	Number of muJava Mutants	Percentage
Cal	6	136	4%
Prime	4	72	6%
TestPat	8	95	8%
TriType	22	200	11%
A1	6	38	16%
A2	9	50	18%
A3	9	46	20%

Program	Number of TRF-TIF Logic Mutants	Number of muJava Mutants	Percentage
A4	40	93	43%
A5	6	40	15%
A6	6	40	15%
A7	5	35	14%
A8	28	60	47%
C1	3	10	30%
C2	6	82	7%
C3	6	82	7%
C4	2	40	5%
C5	5	8	63%
C6	4	82	5%
C7	4	82	5%
C8	3	46	7%
C9	3	37	8%
C10	7	56	13%
C11	2	1	200%
C12	4	3	133%
C13	2	6	33%
C14	4	8	50%
C15	3	14	21%
C16	8	135	6%
C17	6	61	10%
C18	15	82	18%
Sum	236	1740	14%

The key result is that the number of TRF-TIF mutants was 14% of the muJava mutants (**Contribution 4a Part I**). However, recall that this comparison is between the number of TRF-TIF mutants (which are solely logic mutants) and the total number of muJava mutants (which are both logic and non-logic mutants). This is still an interesting comparison because if tests that weakly kill all TRF-TIF mutants kill a vast majority of muJava mutants, the comparison shows that mutant set size can be significantly reduced with little impact on mutation score by solely generating TRF-TIF

mutants. A total of 236 TRF-TIF logic mutants were generated for the 30 programs (with a low of 2, a high of 40 and an average of 7.87). A minimal test set of 143 tests was used to weakly kill these mutants (with a low of 1, a high of 34 and an average of 4.77). muJava generated 1740 mutants for the 30 programs (with a low of 1, a high of 200 and an average of 58).

The results in Table 38 can be explained by a number of factors, related to the fact that the mutation operators in TRF-TIF logic mutation and muJava are different. For the TRF-TIF tool, the logic portion of a Java program is restricted to predicates in “for”, “while”, “if” or “else if” statements. muJava differs in that it possesses non-logic mutation operators and the logic mutation operators it does possess are different than those in TRF-TIF mutation. muJava applies its logic mutation operators to any logic expression in a program and these logic mutation operators do not take advantage of the fault hierarchy. In other words, logic mutants are produced by muJava that are guaranteed to be killed by tests killing other logic mutants.

To compare mutant set size between TRF-TIF mutation and muJava based solely on the number of logic mutants that muJava generates, a side empirical study was conducted. In this study the 19 TCAS Boolean predicates in Appendix B were used as the source under test. TRF-TIF mutants and muJava logic mutants were generated for each predicate and mutant set size was compared. When considering individual minimal DNF predicates, the logic mutation operators in muJava will replace each AND with both XOR and OR and each OR with both AND and XOR. Furthermore, the logic mutation operators in muJava will both insert and delete a negation operator at every possible

location. As an example, consider the predicate $a + b + c$. muJava will generate these mutations: $ab + c$, $a \text{ XOR } b + c$, $a + bc$, $a + b \text{ XOR } c$, $!a + b + c$, $a + !b + c$, $a + b + !c$, $!(a + b) + c$, $a + !(b + c)$ and $!(a + b + c)$. Table 39 shows the results. The key result is that the number of TRF-TIF mutants was 25% of the number of muJava logic mutants (Contribution 4a Part II).

Table 39 TRF-TIF Mutant Set Size vs. muJava Logic Mutant Set Size

Predicate	TRF-TIF [19]	muJava Logic	Percentage	2^n
1	27	102	26.47%	128
2	81	404	20.05%	512
3	148	761	19.45%	4096
4	9	24	37.50%	32
5	34	126	26.98%	512
6	62	193	96.88%	2048
7	62	214	32.12%	1024
8	36	104	34.62%	256
9	16	43	37.21%	128
10	62	199	31.16%	8192
11	61	232	26.29%	8192
12	N/A	N/A	N/A	N/A
13	17	58	29.31%	4096
14	22	67	32.84%	128
15	39	160	24.38%	512
16	104	535	19.44%	4096
17	39	115	33.91%	2048
18	48	148	32.43%	1024
19	16	68	23.54%	256
20	14	37	37.84%	128
Sum	897	3590	24.99%	37,408
Avg	47.21	188.94	24.99%	1968.84

9.2 Equivalent Mutant Set Size (Contribution 4b)

The TRF-TIF approach prevents equivalent mutants from being created assuming reachability, propagation and a complete Boolean space. The muJava tool can generate equivalent mutants even if these assumptions hold. In the empirical study, the number of strongly equivalent mutants generated by each tool was determined manually for each of the 30 programs. Results are displayed in Table 40.

Table 40 Number and Percentage of Strongly Equivalent Software Mutants [23]

Program	Number of Strongly Equivalent TRF-TIF Mutants	Percent of TRF-TIF Mutants that are Strongly Equivalent	Number of Strongly Equivalent muJava Mutants	Percent of muJava Mutants that are Strongly Equivalent
Cal	1	17%	25	18%
Prime	0	0%	5	7%
TestPat	0	0%	6	6%
TriType	0	0%	44	22%
A1	0	0%	6	16%
A2	0	0%	0	0%
A3	0	0%	0	0%
A4	0	0%	0	0%
A5	0	0%	2	5%
A6	0	0%	2	5%
A7	0	0%	10	29%
A8	0	0%	2	3%
C1	3	100%	8	80%
C2	0	0%	8	10%
C3	0	0%	8	10%
C4	0	0%	0	0%
C5	3	60%	6	75%
C6	0	0%	8	10%
C7	0	0%	8	10%
C8	3	100%	9	20%
C9	3	100%	7	19%
C10	6	86%	12	21%

Program	Number of Strongly Equivalent TRF-TIF Mutants	Percent of TRF-TIF Mutants that are Strongly Equivalent	Number of Strongly Equivalent muJava Mutants	Percent of muJava Mutants that are Strongly Equivalent
C11	0	0%	0	0%
C12	0	0%	0	0%
C13	0	0%	1	17%
C14	0	0%	1	13%
C15	3	100%	10	71%
C16	1	13%	3	2%
C17	1	17%	7	11%
C18	3	20%	7	9%
Sum	27	11%	205	12%

The key result is that the number of strongly equivalent TRF-TIF mutants was 13% of the number of strongly equivalent muJava mutants (27 vs. 205) (**Contribution 4b**). However, recall that this comparison is between the number of strongly equivalent TRF-TIF mutants (which are solely logic mutants) and the total number of strongly equivalent muJava mutants (which are both logic and non-logic mutants). The TRF-TIF tool generated one weakly equivalent mutant, as infection was feasible for all but one predicate. For that predicate, the combination of literal values needed to cause infection was infeasible. (If a tester specifies the infeasible combinations of values for the literals in this predicate, then the TRF-TIF tool generates no weakly equivalent mutants. See Appendix J for how these infeasible combinations are specified in the code.) However, 26 of the TRF-TIF logic mutants were strongly equivalent, although not weakly equivalent. For these mutants, infection was achieved but not propagation. This was due to sorting and searching methods in the Collections class that used predicates solely for

improving efficiency. In other words, whether these predicates evaluated to TRUE or FALSE did not impact the output itself, but rather how quickly the output was returned. The author examined the muJava mutants manually and determined that 205 of them were equivalent based on strong mutation.

9.3 Single Minimal DNF Fault Detection (Contribution 4c)

This section presents a theoretical contribution by proving that tests that weakly kill all muJava mutants are guaranteed to detect 5 of the 9 single fault types in Lau and Yu's fault hierarchy.

Theorem 8 (Contribution 4c): TRF-TIF vs. muJava Single Minimal DNF Fault Detection

For software containing minimal DNF predicates, tests that weakly kill all muJava mutants are guaranteed to detect 5 of the 9 single fault types in Lau and Yu's fault hierarchy.

Proof:

When considering individual minimal DNF predicates, the logic mutation operators in muJava will replace each AND with both XOR and OR and each OR with both AND and XOR. Furthermore, the logic mutation operators in muJava will both insert and delete a negation operator at every possible location. As an example, consider the predicate $a + b + c$. muJava will generate these mutations: $ab + c$, $a XOR b + c$, $a + bc$, $a + b XOR c$, $!a + b + c$, $a + !b + c$, $a + b + !c$, $!(a + b) + c$, $a + !(b + c)$ and $!(a + b + c)$.

A test set that kills all muJava mutants is guaranteed to detect the ORF+, LNF, ORF., TNF and ENF. To prove this, it is sufficient to prove that the ORF+, LNF and ORF. are detected because the TNF and ENF follow based on the fault hierarchy. A test set that kills all muJava mutants is guaranteed to kill an ORF+ because one of the mutation operators is to replace OR with AND. Likewise, a test set that kills all muJava mutants is guaranteed to kill an ORF. because one of the mutation operators is to replace AND with OR. Finally, a test set that kills all muJava mutants is guaranteed to kill an LNF because one of the mutation operators is to insert a negation before each literal and another mutation operator deletes a negation before each literal.

A test set that kills all muJava mutants is not guaranteed to detect an LIF, TOF, LRF, or LOF. To prove this, it is sufficient to find a predicate, create muJava mutations for the predicate and show how each fault can go undetected by a test set killing the muJava mutants. First, consider the LOF. AN LOF for literal b in the predicate ab would result in a faulty predicate of a . The only test that detects this fault is 10, which causes predicate ab to evaluate to FALSE and predicate a to evaluate to TRUE. Note that 10 is the only NFP for literal b in predicate ab . muJava will generate the following mutations for predicate ab : $a + b$, $a \text{ XOR } b$, $!ab$, $a!b$ and $!(ab)$. A test set of {01, 11} kills these four mutants but does not include the test 01 and thus does not include an NFP for literal b . When a test set does not include an NFP for a literal, an LOF for that literal is guaranteed not to be detected [30]. Next, consider the LIF, LRF and TOF with an example predicate of $ab + cd$. Table 41 describes the muJava mutants and a test that kills each mutant.

Table 42 describes an LIF, LRF and TOF that can go undetected by a test set that kills all the muJava mutants.

Table 41 muJava Mutants for Predicate $ab + cd$

Mutant	A Test That Kills the Mutant	Value of Original Predicate	Value of Mutated Predicate
$a + b + cd$	1000	FALSE	TRUE
$a \text{ XOR } b + cd$	1000	FALSE	TRUE
$abcd$	0011	TRUE	FALSE
$ab \text{ XOR } cd$	1111	TRUE	FALSE
$ab + c + d$	0001	FALSE	TRUE
$ab + c \text{ XOR } d$	0001	FALSE	TRUE
$!ab + cd$	0100	FALSE	TRUE
$a!b + cd$	1000	FALSE	TRUE
$ab + !cd$	0001	FALSE	TRUE
$ab + c!d$	0010	FALSE	TRUE
$!(ab) + cd$	0000	FALSE	TRUE
$ab + !(cd)$	0000	FALSE	TRUE
$!(ab + cd)$	0000	FALSE	TRUE

Table 42 LIF, LRF and TOF for Predicate $ab + cd$

Fault	All Tests Detecting the Fault	Value of Original Predicate	Value of Mutated Predicate
$abc + cd$ (LIF)	1101, 1100	TRUE	FALSE
$ac + cd$ (LRF)	1101, 1100, 1010	TRUE for 1101, 1100 FALSE for 1010	FALSE for 1101, 1100 TRUE for 1010
cd (TOF)	1100, 1101, 1110	TRUE	FALSE

The test set formed by the union of all the tests in the second column in Table 41 does not include any of the tests in the second column of Table 42. Thus, the LIF, LRF and TOF go undetected. Note that the test set formed by the union of all the tests in the

second column in Table 41 does not include a UTP for term ab . The only UTPs for term ab are 1100, 1101 and 1110. When a test set does not include a UTP for a term, both a TOF for that term and an LIF in that term are guaranteed not to be detected [30]. It is usually possible to detect an LRF for a literal in a term without including a UTP for that term [30] and in this case the NFP 1010 detects the LRF in Table 42, but this point does not need to be included in a test set killing all muJava mutants.

End Proof

9.4 Double Minimal DNF Fault Detection (Contribution 4d)

This section presents a theoretical contribution by proving that tests that weakly kill all muJava mutants are guaranteed to detect less double fault types in Lau and Yu's fault hierarchy than Minimal-MUMCUT tests.

Theorem 9 (Contribution 4d): TRF-TIF vs. muJava Double Minimal DNF Fault Detection

For software with minimal DNF predicates, tests that weakly kill all muJava mutants are guaranteed to detect fewer double fault types in Lau and Yu's fault hierarchy than Minimal-MUMCUT tests.

Proof:

Kaminski and Ammann [20] show that tests satisfying the Minimal-MUMCUT criterion (and hence killing all TRF-TIF logic mutants) guarantee detecting 84 of the 92 double fault types. Any test set that includes a UTP for each term and an NFP for each literal is guaranteed this double fault detection [27, 28, 29]. A test set that kills all muJava mutants

guarantees detection of fewer double faults because it is not guaranteed to include a UTP for each term and an NFP for each literal as described in section 9.3.

End Proof

The exact number of double faults guaranteed to be detected by a test set that kills all muJava mutants is not known. What makes the analysis complex is that a test set that kills all muJava mutants does make some guarantees regarding UTPs and NFPs. Such a test set is guaranteed to contain a UTP for at least one of every two adjacent terms as this is required to detect the ORF+ [30]. Such a test is also guaranteed to contain either a UTP for a term or an NFP for every literal in a term as this is required to detect the LNF [30].

For an example, the TOF-TOF double fault is guaranteed to be detected by a Minimal-MUMCUT test set [30]. Examining Table 41 shows that a test set that kills all muJava mutants includes only one test that makes term ab TRUE, namely 1111, which also makes term cd TRUE. Extending this further, consider predicate $ab + cd + ef + gh$. The only muJava mutant that requires term ab to be TRUE is changing the first OR to XOR ($ab \text{ XOR } cd + ef + gh$). To kill the corresponding mutant, a test must make both ab and cd TRUE while making both ef and gh FALSE. 11110000 is such a test. The only muJava mutant that requires term gh to be TRUE is changing the last OR to XOR ($ab + cd + ef \text{ XOR } gh$). To kill the corresponding mutant, a test must make both ef and gh TRUE while making both ab and cd FALSE. 00001111 is such a point. Lau, Liu and Yu [27, 28, 29] state that to detect the TOF-TOF, any point that makes either (or both) terms being omitted TRUE while making all other terms FALSE will detect the fault. Consider the TOF-TOF where both terms ab and gh are omitted to yield $cd + ef$. Neither 11110000

nor 00001111 detect this fault because both points make some other term besides *ab* or *gh* TRUE.

9.5 General Fault Detection (Contribution 4f)

Using muJava as the fault seeding tool, the study described at the beginning of this chapter examined general fault detection of a test set that weakly kills all TRF-TIF logic mutants. Specifically, the author captured the percentage of strongly non-equivalent muJava mutants that were strongly killed by a test set that weakly kills all TRF-TIF logic mutants. Results are shown in Table 43.

Table 43 Number of Strongly non-Equivalent muJava Mutants Strongly Killed by a Test Set that Weakly Kills All TRF-TIF Logic Mutants [23]

Program	Number of TRF-TIF Logic Mutants	Number of Strongly Killed muJava Mutants	Number of Strongly Non-Equivalent muJava Mutants	Percentage
Cal	6	106	111	96%
Prime	4	66	67	99%
TestPat	8	78	89	88%
TriType	22	153	156	99%
A1	6	31	32	97%
A2	9	45	50	90%
A3	9	40	46	87%
A4	40	87	93	94%
A5	6	38	38	100%
A6	6	38	38	100%
A7	5	25	25	100%
A8	28	58	58	100%
C1	3	2	2	100%
C2	6	68	74	92%
C3	6	68	74	92%
C4	2	34	40	85%
C5	5	2	2	100%
C6	4	68	74	92%
C7	4	68	74	92%

Program	Number of TRF-TIF Logic Mutants	Number of Strongly Killed muJava Mutants	Number of Strongly Non-Equivalent muJava Mutants	Percentage
C8	3	37	37	100%
C9	3	30	30	100%
C10	7	37	44	84%
C11	2	1	1	100%
C12	4	3	3	100%
C13	2	5	5	100%
C14	4	7	7	100%
C15	3	4	4	100%
C16	8	126	132	95%
C17	6	51	54	94%
C18	15	61	75	81%
Sum	236	1437	1535	94%

The TRF-TIF test set of 143 tests strongly killed 94% (1437 / 1535) of the strongly non-equivalent muJava mutants (**Contribution 4f**), with a low of 81%, a high of 100% and a standard deviation of 6%.

The 30 programs selected for the initial empirical study were small. To see how the findings scale to larger program, a calculator Open Source Software Java program was selected for a second empirical study. The calculator program had 351 statements, 51 logic statements and 62 unique literals. The calculator program outputs to a console the values of its variables after a user pressed a button on the calculator user interface. This allowed weak mutation testing to be applied since the program state could be known after executing a test.

A total of 95 TRF-TIF logic mutants were generated for the calculator program, none of which were weakly equivalent. A minimal test set of 39 tests was used to weakly

kill these mutants. This test set is minimal in the sense that if even one of the 39 tests is removed from the test set, at least one non-equivalent mutant cannot be weakly killed. The muJava tool generated 767 mutants for the calculator program. The number of strongly equivalent muJava mutants was not determined due to the large number of muJava mutants generated. However, Offutt [32] reports that in programs he examined, 10% of the mutants were strongly equivalent. Also, 12% of the muJava mutants were strongly equivalent for the 30 small programs in the initial study. A 10% estimate means that there are 690 strongly non-equivalent muJava mutants. The 39 tests weakly killed 642 muJava mutants (93%), indicating that results scaled to a larger program.

Threats to Validity and Sources of Bias

The percentage of muJava mutants strongly killed by a test set that weakly kills all TRF-TIF mutants will depend on how “predicate heavy” the program under test is. This is a threat to external validity because it limits the generalizability of the results to programs that are as “predicate heavy” as the programs in the empirical study. To formalize this, two new terms are introduced, the *Logic Statement Ratio* and the *Unique Literals Ratio*.

Logic Statement Ratio is defined as the ratio of the number of logic statements to the total number of statements. A logic statement is considered to be an “if” or “else if” predicate and the number of statements is counted as the number of statements ending in a semicolon (excluding package and import statements). For the “if” and “else if” predicates as a whole for all 30 programs, the average Logic Statement Ratio was 0.22 with a low of 0.07, a high of 0.53 and a standard deviation of 0.13. The Logic Statement

Ratio does not take into account the number of unique literals in a predicate. Thus, the Unique Literals Ratio is defined as the ratio of the total number of unique literals in all logic statements to the total number of statements. For the “if” and “else if” predicates as a whole for all 30 programs, the average Unique Literals Ratio was 0.28 with a low of 0.11, a high of 0.93 and a standard deviation of 0.16. Based on this data, the author suggests that (assuming unique literals are spread evenly throughout the source code) a conservative estimate is that a test set that weakly kills all TRF-TIF mutants will strongly kill at least 80% of strongly non-equivalent muJava mutants when the Unique Literals Ratio is greater than 0.10.

The studies undertaken are subject to at three sources of bias. The authors considered a logic statement to be an “if” or “else if” predicate. Predicates in “while” and “for loops” were not considered as logic statements (although the TRF-TIF tool has an option to generate mutants based on loop predicates). In general, omitting loop predicates creates an experimental bias against the ability of a test set that weakly kills all TRF-TIF logic mutants to kill non-logic mutants. However, for all but one of the programs studied, tests used to weakly kill all TRF-TIF logic mutants based on “if” and “else if” predicates were found to weakly kill all TRF-TIF logic mutants based on mutations to “while” and “for loop” predicates. Thus, additional tests cases would be needed for only one program to weakly kill all TRF-TIF logic mutants had loop predicates been included. Hence, the ability of a TRF-TIF test set to kill non-logic mutants for 29 of the programs would not have changed, meaning the bias created by using only “if” and “else if” predicates in TRF-TIF logic mutations is minimal.

Another source of bias is that the way the authors classified a mutant as equivalent. Recall that classifying a TRF-TIF logic mutant as equivalent means that it is equivalent based on weak mutation testing, which means that it will always be equivalent based on strong mutation testing. However, classifying a muJava mutant as equivalent means it is equivalent under strong mutation testing, although it might not be equivalent under weak mutation testing. In general, this will create an experimental bias against TRF-TIF logic mutation because it requires that tests that weakly kill TRF-TIF logic mutants strongly kill non-logic muJava mutants. However, usually test that weakly kill mutants also strongly kill mutants, so this source of bias is likely to have little impact.

A third of source of bias is that muJava is itself a selective mutation tool in that it uses a subset of common mutation operators that have been shown to be highly effective [2]. Thus, the comparison in this study is between TRF-TIF logic mutation operators with a selective mutation operator set. Thus, the mutation score of 94% is likely to be smaller when replacing the selective mutation operators of muJava with a full set.

One interesting finding of the study was the variance seen in the results. While for the software empirical evaluation, an average of 94% of strongly non-equivalent muJava mutants were killed by a test that weakly killed all weakly non-equivalent TRF-TIF mutants, this percentage was as low as 81% for one of the programs. What is interesting is that the software program for which TRF-TIF logic mutation scored the lowest (81% for the Collections replaceAll method), had a slightly above average Unique Literals Ratio (0.29 compared to an average of 0.28). In fact, the correlation between the Unique Literals Ratio and the percentage of strongly non-equivalent muJava mutants killed by a

test that weakly killed all weakly non-equivalent TRF-TIF logic mutants was very weak ($r = 0.03$). Future research is planned to investigate why the percentages of general mutants killed by tests weakly killing all weakly non-equivalent TRF-TIF logic mutants were lower for some programs than others. Based on our data, a Unique Literals Ratio of at least 0.10 is a reliable indicator for achieving at least an 80% mutation score, but other factors influence whether the mutation score goes higher.

10 Comparison of TRF-TIF Logic Mutation with SQLMutation

This chapter is the last of three comparing the TRF-TIF logic mutation approach with other mutation tools/approaches. The focus of this chapter is comparing the TRF-TIF logic mutation approach with SQLMutation, which corresponds to the fifth row in Table 5. SQLMutation is an online query mutation tool based on CACC. To kill all SQLMutation mutants, a criterion known as SQLFpc (SQL Full predicate coverage) must be satisfied, which itself requires that CACC be satisfied. Each section in this chapter corresponds to a cell in Table 5. Section 10.1 corresponds to cell 5a (mutant set size comparison), section 10.2 corresponds to cell 5b (equivalent mutant set size comparison), section 10.3 corresponds to cell 5c (single minimal DNF fault detection comparison), section 10.4 corresponds to cell 5d (double minimal DNF fault detection comparison), and section 10.5 corresponds to cell 5e (general fault detection). The overriding theme of this chapter is that TRF-TIF logic mutation provides better minimal DNF fault detection and reduced mutant set size when compared to SQLMutation, and that TRF-TIF logic mutation is effective at producing mutants that when killed, also kill a high percentage of non-logic mutants. The rest of this section describes an empirical study from which contributions in subsequent sections are derived.

A database empirical evaluation [23] was conducted to ascertain if logic mutation testing can be used to reduce the costs of general mutation testing while maintaining most of its benefits. Specifically, the study was performed to answer these questions:

- 1) Does TRF-TIF logic mutation produce fewer mutants than the SQLMutation approach with minimal impact on general fault detection?
- 2) Does TRF-TIF logic mutation produce fewer mutants than the SQLMutation approach while improving logic fault detection?
- 3) Does TRF-TIF logic mutation produce fewer equivalent mutants than the SQLMutation approach?
- 4) Can TRF-TIF logic mutation reduce test set size (the number of rows needed in the tables of a database to kill all mutants) as compared to the SQLFpc approach while still maintaining a high mutation score?
- 5) How do the mutation scores for TRF-TIF logic mutation compare with an approach where a test set is created randomly?

Ten queries from an open source project called Compiere were used. These queries were selected because Tuya et al. [42] used views from the Compiere project to compare mutation scores for the SQLFpc approach with an approach using a database populated randomly.

Kaminski et al. [23] examined all the views in the Compiere project and extracted from them any query or sub query that contained a predicate (meaning a condition in a JOIN or a clause in a CASE, WHERE or HAVING statement) that had at least three unique literals. There were ten such queries. The rationale is that an approach based on

logic mutation is going to require predicates with at least three unique literals to be beneficial. For six of the ten queries, the query constitutes the entire view. For the other four queries, the query consisted of a sub query in the view. The study used the TRF-TIF tool to generate logic mutants and version 1.2.59 of the SQLMutation tool to generate general mutants. In some cases the queries were modified so as to eliminate internal PL/SQL functions or to get them to run successfully through the SQLMutation tool but in no cases were any join conditions changed and in no cases were the number of unique literals in a predicate changed.

All predicates were already in minimal DNF for 8 of the 10 queries. For query 4, the WHERE clause predicate was in minimal CNF. For query 1, the WHERE clause predicate was neither in minimal DNF nor minimal CNF. For these two queries, the WHERE clause predicate was converted to minimal DNF before generating the TRF-TIF mutants. When a query had multiple predicates (for example, a WHERE clause and a JOIN condition), the TRF-TIF approach was applied if the predicate had at least three unique literals whereas a combinatorial approach was applied if the predicate had less than three unique literals. When creating test data to kill all TRF-TIF mutants, the author had not seen the SQLMutation mutants to eliminate bias. Appendix K lists the following for each query:

- 1) The schema used for the purpose of running the SQLMutation tool
- 2) The actual SQL for the query
- 3) The main WHERE clause predicate in minimal DNF
- 4) The mutants created by the TRF-TIF tool for the main WHERE clause predicate

- 5) The test points needed to kill each mutant in terms of literal values
- 6) The test points needed to kill all the TRF-TIF mutants in terms of the rows needed in a test database

10.1 Mutant Set Size (Contribution 5a)

The number of mutants generated by each tool was determined and the results are in Table 44. The key finding is that TRF-TIF mutant set size is 2% of SQLMutation mutant set size (Contribution 5a).

Table 44 Number of Query Mutants [23]

Query	TRF-TIF Logic Mutants	SQLMutation Mutants*	Percentage
1	71	1406	5.05%
2	5	1007	0.50%
3	5	1005	0.50%
4	20	1025	1.95%
5	4	49	8.16%
6	4	252	1.59%
7	4	497	0.80%
8	5	401	1.25%
9	5	610	0.82%
10	4	166	2.41%
Sum	127	6418	1.98%

*The SQLMutation tool can automatically identify a few equivalent mutants. (9 of the 6427 total mutants generated for the 10 queries were identified as equivalent). The data in Table 44 reflects only the mutants generated by the SQLMutation tool that the SQLMutation tool does not mark as equivalent.

These results can be explained by the fact that the TRF-TIF tool produces different logic mutants than the logic mutants produced by SQLMutation as the logic mutation operators in the TRF-TIF tool are different than the logic mutation operators in SQLMutation. For the TRF-TIF tool, the logic portion of a query is considered to be a WHERE, HAVING or CASE clause or a JOIN condition. The TRF-TIF approach limits itself to these conditions and clauses because these are the places where a predicate is explicitly specified. The SQLFpc approach is more comprehensive. Thus, less data in a test database is needed for the TRF-TIF approach than for the SQLFpc approach as fewer mutants are generated. Another explanation is that the TRF-TIF approach avoids generating unnecessary mutants low in the fault hierarchy. SQLMutation does not take advantage of the fault detection relations in the fault hierarchy. A final explanation is that SQLMutation generates more equivalent mutants, which is described next.

10.2 Equivalent Mutant Set Size (Contribution 5b)

The SQLMutation tool automatically detects some equivalent mutants. However, it also produces equivalent mutants that are not detected. The author of the SQLMutation tool was asked for an estimate as to what percentage of the mutants generated by the SQLMutation tool are equivalent, yet are not detected as such. At the time of the correspondence, the current version of the SQLMutation tool was version 1.2.59. The author of the SQLMutation tool indicated that while the percentage varies depending on the complexity of the query, a conservative estimate is 6%. This was based on an NIST study using simple queries. More complex queries are likely to have a higher percentage of equivalent mutants generated by the SQLMutation tool. Thus, a TRF-TIF mutant set

contains fewer equivalent mutants than a SQLMutation mutant set (**Contribution 5b**) since a TRF-TIF mutant set is guaranteed to not have any equivalent query mutants (assuming a complete Boolean space) and the 6% estimate for SQLMutation is based on a complete Boolean space.

10.3 Single Minimal DNF Fault Detection (Contribution 5c)

This section presents a theoretical contribution by proving that tests that weakly kill all SQLMutation mutants are guaranteed to detect 2 of the 9 single fault types in Lau and Yu's fault hierarchy (the TNF and ENF).

Theorem 10 (Contribution 5c): TRF-TIF vs. SQL Mutation Single Minimal DNF Fault Detection

For queries having minimal DNF WHERE clauses, tests that weakly kill all SQLMutation mutants are guaranteed to detect 2 of the 9 single fault types in Lau and Yu's fault hierarchy.

Proof:

SQLFpc is based on masking MCDC (CACC) which is known to guarantee detection of only 2 of the 9 faults in Lau and Yu's fault hierarchy [24]. (CACC requires a UTP and NFP for each unique literal but not each literal). This was also proven in section 6.2 by showing how RACC tests guarantee detecting only 2 of the faults because a CACC test is a RACC test set.

End Proof

10.4 Double Minimal DNF Fault Detection (Contribution 5d)

This section presents a theoretical contribution by proving that tests that weakly kill all SQLMutation mutants are guaranteed to detect 22 of the 92 double fault types in Lau and Yu's fault hierarchy.

Theorem 11 (Contribution 5d): TRF-TIF vs. SQL Mutation Double Minimal DNF Fault Detection

For queries having minimal DNF WHERE clauses, tests that weakly kill all SQLMutation mutants are guaranteed to detect 22 of the 92 double fault types in Lau and Yu's fault hierarchy.

Proof:

SQLFpc is based on masking MCDC (CACC) which is known to guarantee detection of only 22 of the 92 double fault types in the hierarchy [23]. (CACC requires a UTP and NFP for each unique literal but not each literal). This was also proven in section 6.3 by showing how RACC tests guarantee detecting only 22 of the faults because a CACC test is a RACC test set. The 22 double fault types correspond only to 4 of the 45 unordered, non-versioned double fault types. The only double fault types that MCDC is guaranteed to detect (ENF-TOF, TNF-TNF, TNF-TOF and TNF-LIF) are those that can be detected by any FALSE point [23].

End Proof

10.5 General Fault Detection (Contribution 5f Parts I and II)

Using SQLMutation as the fault seeding tool, the study described at the beginning of this chapter examined general fault detection of a test set that weakly kills all TRF-TIF logic mutants. The data collected included:

- 1) percentage of non-equivalent* SQLMutation tool mutants killed by a test set killing all TRF-TIF logic mutants
- 2) a comparison of TRF-TIF logic mutation score with mutation score based on populating a test database randomly

*The percentage of equivalent mutants is assumed to be 8% for two of the queries and 6% for all other queries based on data provided by the author of the SQLMutation tool. 8 of the 10 queries had 5 or less unique literals in their WHERE clause. The two remaining queries had 10 and 18 unique literals in their WHERE clauses so the percentage of equivalent mutants for these queries is assumed to be 8%.

Table 45 displays the percentage of non-equivalent SQLMutation mutants weakly killed by a test set that kills all TRF-TIF logic mutants. The data in Table 45 reflects assumptions as to the proportion of equivalent mutants. Based on Table 45, a conservative estimate is that a test set that kills all TRF-TIF mutants will kill at least 80% of non-equivalent SQLMutation mutants when the WHERE clause has at least 3 unique literals

Table 45 Percentage of Non-Equivalent SQLMutation Mutants Killed by a Test Set Killing all TRF-TIF Logic Mutants [23]

Query	Killed Non-Equivalent SQLMutation Mutants	Non-Equivalent SQLMutation Mutants	Percentage
1	1066	1295	82%
2	864	947	91%
3	998	*1005	99%
4	828	967	86%
5	42	46	91%
6	221	237	93%
7	445	468	95%
8	352	379	93%
9	477	573	83%
10	144	156	92%
Sum	5437	6053	90%

* 1005 mutants were generated by the SQLMutation tool for query number 3. Since 998 of these mutants were killed at most 7 of them are equivalent. Thus, for this query, the 6% estimate of equivalent mutants is too high. Thus, it is assumed all 1005 mutants are not equivalent. This assumption biases the results against the TRF-TIF logic mutation tool as some of these 1005 mutants may be equivalent.

Table 46 compares mutation scores for the TRF-TIF approach versus an approach based on populating a test database randomly with four rows of data per table as described by Tuya et al. [42]. Table 46 shows data for queries 1-5 and 9. These queries are targeted because Tuya’s research focused on testing entire views and queries 1-5 and 9 represent the entire view (whereas queries 6, 7, 8 and 10 represent a subquery within the view). Thus, a direct comparison between the TRF-TIF results and random results are only possible for queries 1-5 and 9. On average, a test set that kills all TRF-TIF logic mutants required 3.54 rows of data per table, so this biases the comparison against the

TRF-TIF approach. The mutation score for the TRF-TIF approach in Table 46 is lower for each query than that listed in Table 45. This is because the random mutation scores are based on the set of mutants without accounting for an estimate of non-equivalent mutants. That is, the mutation scores for the random approach assumed an equivalent mutant percentage of 0%. Thus, the actual mutation scores for the TRF-TIF approach and the random approach are higher than what is presented in Table 46. However, not taking into account the assumption of equivalent mutant frequency allows an unbiased comparison between the TRF-TIF approach and the random approach.

Table 46 Mutation Scores for TRF-TIF Logic Mutation versus a Random Approach [23]

Query	TRF-TIF Logic Mutation Score	Random Mutation Score
1	75.82%	1.6%
2	85.80%	0.2%
3	99.30%	0.2%
4	80.78%	0.1%
5	85.71%	18.7%
9	78.20%	0.0%
Average	84.27%	3.47%

Key results include:

- 1) A test set that killed all TRF-TIF mutants killed 90% of all non-equivalent SQLMutation mutants (**Contribution 5f Part I**) with a standard deviation of 5%

- 2) Mutation score for the TRF-TIF approach is more than 20 times higher than that of a random approach, even when the random approach uses more test data (**Contribution 5f Part II**). (Even when a random approach populates 1000 rows per table, the mutation score is still higher for the TRF-TIF approach based on 3.54 rows per table.)

Tuya et al. [42] give detailed results for the SQLFpc approach only for query 1. For this query, they state that 87.30% of SQLMutation mutants are killed by a test set satisfying SQLFpc. They also specify that 126 rows are needed to satisfy SQLFpc coverage. The author contacted Tuya to get similar data for queries 2, 3, 4, 5 and 9. These queries are targeted because Tuya's research focused on testing entire views and queries 1-5 and 9 represent the entire view (whereas queries 6, 7, 8 and 10 represent a subquery within the view). Thus, a direct comparison between the TRF-TIF results and Tuya's results is only possible for queries 1-5 and 9.

Table 47 gives Tuya's data for a test set satisfying SQLFpc for queries 1-5 and 9. The mutation score represents no assumption as to the number of equivalent mutants (only mutants identified as equivalent by the SQLMutation tool are accounted for). Note that the mutation score for query 1 in Table 47 is different than what is mentioned above because Tuya updated his tool to eliminate randomness so as to make results repeatable (meaning the mutant set is always the same for a given query that is run through the SQLMutation tool). Note also that number of rows needed for query 1 in Table 47 is different than what is mentioned above since Tuya indicated that the 126 rows was an

error and it should have been reported as 136. Table 48 gives data for a test set based on TRF-TIF logic mutation for queries 1-5 and 9.

Table 47 Mutation Scores and Number of Database Rows for a SQLFpc Test Set [42]

Query	Mutation Score	Rows
1	86.15%	136
2	89.15%	35
3	91.70%	42
4	90.30%	44
5	94.67%	5
9	55.42%	25
Average	84.57%	47.83

Table 48 Mutation Scores and Number of Database Rows for a TRF-TIF Logic Mutation Test Set [23]

Query	Mutation Score	Rows
1	75.82%	46
2	85.80%	13
3	99.30%	17
4	80.78%	22
5	85.71%	4
9	78.20%	9
Average	84.27%	18.5

The average mutation scores for the SQLFpc approach and the TRF-TIF approach are almost the same. However, the SQLFpc approach requires more than 2.5 times as number of rows in the database. Thus, test set size for the TRF-TIF approach is about

38% of the test set size for the SQLFpc approach. Also, the TRF-TIF approach guarantees detecting faults in Lau and Yu’s fault hierarchy that the SQLFpc approach does not. The SQLMutation tool does not generate mutants that correspond to some of the faults in the hierarchy. For example, there are no mutants generated by the SQLMutation tool that correspond to the LIF. If the SQLMutation tool did generate such mutants, the mutation score for the SQLFpc approach would decrease and the mutation score for the TRF-TIF approach would increase as tests that weakly kill all TRF-TIF mutants guarantee detecting the LIF. Since SQLFpc is based on MCDC and MCDC does not guarantee detecting 7 of the 9 faults in Lau and Yu’s fault hierarchy, a test set that kills all TRF-TIF mutants is guaranteed to detect 7 fault types that a SQLFpc test does not.

Threats to Validity and Sources of Bias

The percentage of SQLMutation mutants killed by a test set that kills all TRF-TIF query mutants will depend on how “predicate heavy” the query under test is. This is a threat to external validity because it limits the generalizability of the results to queries that are as “predicate heavy” as the programs in the empirical study.

The study undertaken has a bias since SQLMutation is itself a selective mutation tool because it uses a subset of query mutation operators. Thus, the comparison in this study is between TRF-TIF mutation operators with a selective mutation operator set. Thus, the mutation score of 90% is likely to be smaller when replacing the selective mutation operators of SQLMutation with a full set.

One interesting finding was the variance seen in the results. For the database empirical evaluation, 90% of non-equivalent SQLMutation mutants were killed by a test set that killed all TRF-TIF logic mutants, but this percentage was as low as 82% for one of the queries. What is interesting is that the query for which TRF-TIF logic mutation scored the lowest (82% for query 1) had more unique literals in its where clause than any other query under test. Also, the query for which TRF-TIF logic mutation scored the second lowest (83% for query 9) had the second most unique literals in its where clause. A negative correlation coefficient of $r = -0.67$ was found between the number of unique literals in the where clause of a query and the number of non-equivalent SQLMutation mutants killed by a test set killing all TRF-TIF logic mutants. Future research is planned to investigate why the percentages of general mutants killed by tests weakly killing all weakly non-equivalent TRF-TIF logic mutants were lower for some queries than others. Based on our data, a query having at least 3 unique literals is a reliable indicator for achieving at least an 80% mutation score, but other factors influence whether the mutation score goes higher.

11 Conclusion

In conclusion, the Minimal-MUMCUT logic coverage criterion and TRF-TIF logic mutation advance the state of software and query testing by providing efficient solutions to the problem of increasing fault detection while decreasing mutant and test set size.

The Minimal-MUMCUT criterion has been shown to both reduce test set size and increase fault detection when compared to other logic coverage criteria, both in theory and in practice. Based on criterion feasibility of individual terms and literals in a minimal DNF or minimal CNF predicate, Minimal-MUMCUT tests can reduce MUMCUT test set size while at the same time detecting more faults than semantic ACC and ICC tests. An evaluation of safety-critical software, open source software and open source queries showed that the majority of predicates are in either minimal CNF or minimal DNF (or both). However, when a predicate is not in minimal CNF or minimal DNF, the Minimal-MUMCUT criterion still provides excellent fault detection. The benefits of single fault detection extend to double fault detection for the Minimal-MUMCUT criterion and in practice, Minimal-MUMCUT test sets were found to detect all but one double fault type. Also, the Minimal-MUMCUT criterion can be extended with just a few tests to guarantee detecting this double fault type.

By using an extended fault hierarchy and the concepts of semantic and syntactic fault size, TRF-TIF logic mutation was shown to have some distinct advantages over

current software mutation and query mutation approaches. Reducing logic mutant set size, reducing database test set size needed to kill mutants, reducing (or eliminating) equivalent mutants, generating more highly selective mutants and helping the tester create test data to kill mutants are all advantages of the TRF-TIF logic mutation approach. It was also shown that a test set that kills all TRF-TIF logic mutants kills a high percentage of mutants in general, both for software and queries. These benefits are the result of using new highly selective logic mutation operators that change the way researchers should view logic mutation testing.

My main logic coverage criterion recommendation is that the Federal Aviation Administration require the Minimal-MUMCUT criterion instead of MCDC for any predicates in minimal DNF or minimal CNF (which comprise the majority of predicates in safety-critical avionics software). The Minimal-MUMCUT criterion provides better logic fault detection for such predicates. Furthermore, it is my recommendation that testers currently using the MUMCUT criterion switch to using the Minimal-MUMCUT criterion. The Minimal-MUMCUT criterion offers the same fault detection as the MUMCUT criterion for an important class of logic faults, and has been shown to detect over 98% of faults in predicates of any syntactic format, as well as an average of 95% of general faults (logic and non-logic faults). Thus, the extra tests required by the MUMCUT criterion are of little, if any, value based on the theoretical and empirical studies conducted in this research.

My main logic mutation recommendation is that mutation tools should undergo a fundamental redesign in terms of the logic mutation operators used. These tools should

apply TRF-TIF mutation operators to minimal DNF predicates instead of the currently used logic mutation operators. Ideally, such tools would also convert non-minimal DNF predicates into minimal-DNF predicates and apply TRF-TIF mutation operators on the converted predicates. Furthermore, in cases where (1) the software under test has a high degree fault tolerance (such as non-safety-critical software) and (2) testing resources are limited, testers should consider using the TRF-TIF mutation tool instead of a tool that generates both logic and non-logic mutants. The reason is that generating only TRF-TIF logic mutants has been shown to significantly reduce mutant set size while maintaining high mutation scores.

Several possibilities exist in terms of future work. One is to repeat the same experiments conducted herein with larger samples of different software programs and queries. Increasing both sample size and diversifying the types of programs and queries under test will strengthen the generalizability of the findings. A second idea for future work is to implement TRF-TIF mutation in current mutation tools such muJava and SQLMutation. A third idea is to examine what other mutation operators besides TRF-TIF mutation operators are needed to bring all the mutation scores of the test sets described in Table 43 and Table 45 to over 95%. That is, it would be interesting to determine what types of general mutants are not killed by a test set that weakly kills all TRF-TIF mutants. In other words, the goal would be to determine empirically what general fault types tend to go undetected by tests that weakly kill all TRF-TIF mutants and then to determine which mutation operators seeded those faults. Supplementing TRF-TIF mutation operators with these additional mutation operators could prove to be a way to add only a

small set of mutation operators to the TRF-TIF mutation operator set while achieving higher mutation scores across the board.

Appendix A Optimization Model for Selecting NFPs

(From section 4.1)

This appendix shows how overlapping NFPs can be modeled as an optimization problem as part of the Minimal-MUMCUT test generation algorithm.

Given a minimal DNF predicate of $ab + cd$ the following NFPs exist:

NFPs for a: 0100, 0101, 0110

NFPs for b: 1000, 1001, 1010

NFPs for c: 0001, 0101, 1001

NFPs for d: 0010, 0110, 1010

The optimization model is

Minimize $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$ subject to

$$x_1 + x_2 + x_3 \geq 1$$

$$x_3 + x_4 + x_6 \geq 1$$

$$x_7 + x_2 + x_5 \geq 1$$

$$x_8 + x_3 + x_6 \geq 1$$

where

x_1 is 1 if 0100 is selected and 0 if it is not selected

x_2 is 1 if 0101 is selected and 0 if it is not selected

x_3 is 1 if 0110 is selected and 0 if it is not selected

x_4 is 1 if 1000 is selected and 0 if it is not selected

x_5 is 1 if 1001 is selected and 0 if it is not selected

x_6 is 1 if 1010 is selected and 0 if it is not selected

x_7 is 1 if 0001 is selected and 0 if it is not selected

x_8 is 1 if 0010 is selected and 0 if it is not selected

There are two optimal solutions:

$$x_2=1, x_6=1 \text{ (all others = 0)}$$

$$x_3=1, x_5=1 \text{ (all others = 0)}$$

Appendix B Minimal DNF TCAS Predicates

Reprinted [46]

1. $a\sim bd\sim e\sim h\sim f + a\sim b\sim de\sim h\sim f + a\sim bcd\sim e\sim f + a\sim bc\sim de\sim f + \sim ab\sim de\sim f$
2. $a\sim bc\sim d\sim e\sim gh\sim i\sim f + a\sim b\sim d\sim e\sim g\sim h\sim if + a\sim b\sim c\sim e\sim g\sim h\sim if + a\sim b\sim c\sim d\sim g\sim h\sim if +$
 $a\sim bc\sim d\sim eg\sim h\sim f + a\sim bc\sim d\sim e\sim hi\sim f + a\sim b\sim cd\sim eg\sim h\sim f + a\sim b\sim cd\sim e\sim hi\sim f + a\sim b\sim c\sim deg\sim h\sim f$
 $+ a\sim b\sim c\sim de\sim hi\sim f + \sim abc\sim d\sim e\sim hi\sim f + \sim ab\sim cd\sim e\sim hi\sim f + \sim ab\sim c\sim de\sim hi\sim f$
3. $\sim a\sim bc\sim g\sim i\sim k\sim m + \sim a\sim bcg\sim h\sim l\sim m + \sim a\sim bc\sim g\sim hi\sim m + \sim a\sim bcgi\sim l\sim m + \sim a\sim bcgi\sim k\sim m +$
 $\sim a\sim bc\sim h\sim k\sim m + \sim ab\sim c\sim g\sim i\sim k + a\sim b\sim c\sim g\sim i\sim k + \sim a\sim bc\sim i\sim kf + \sim ab\sim c\sim g\sim hi +$
 $\sim ab\sim cg\sim h\sim l + a\sim b\sim c\sim g\sim hi + a\sim b\sim cg\sim h\sim l + \sim a\sim bc\sim hif + \sim ab\sim cgi\sim k + \sim ab\sim cgi\sim l +$
 $a\sim b\sim cgi\sim k + a\sim b\sim cgi\sim l + a\sim b\sim c\sim h\sim k + \sim ab\sim c\sim h\sim k + a\sim b\sim cgf + \sim ab\sim cgf + \sim a\sim bcgf +$
 $a\sim b\sim c\sim d + a\sim b\sim c\sim e$
4. $a\sim bd + a\sim cd + e$
5. $a\sim g\sim i\sim k + ag\sim h\sim l + a\sim g\sim hi + agi\sim l + agi\sim k + a\sim h\sim k + a\sim c + a\sim b + f$
6. $\sim ab\sim cdeg\sim hij\sim k\sim f + a\sim bc\sim deg\sim hij\sim k\sim f + \sim ab\sim cde\sim g\sim h\sim jf + \sim ab\sim cde\sim g\sim h\sim kf +$
 $a\sim bc\sim de\sim g\sim h\sim jf + a\sim bc\sim de\sim g\sim h\sim kf$
7. $\sim ab\sim cde\sim g\sim i\sim j + \sim ab\sim cde\sim h\sim i\sim k + a\sim bc\sim de\sim g\sim i\sim j + a\sim bc\sim de\sim h\sim i\sim k + a\sim bc\sim de\sim g\sim k +$
 $a\sim bc\sim de\sim h\sim j + \sim ab\sim cde\sim g\sim k + \sim ab\sim cde\sim h\sim j$
8. $\sim ab\sim cde\sim gh\sim f + a\sim bc\sim de\sim gh\sim f + \sim ab\sim cdeg\sim hf + a\sim bc\sim deg\sim hf$
9. $\sim a\sim b\sim cd\sim e\sim gf + \sim abc\sim d\sim e\sim gf$
10. $a\sim b\sim cd\sim eg\sim j\sim l\sim mf + a\sim b\sim cd\sim eh\sim j\sim l\sim mf + a\sim b\sim cd\sim ei\sim j\sim l\sim mf + a\sim b\sim cd\sim egj\sim k\sim mf +$
 $a\sim b\sim cd\sim ehj\sim k\sim mf + a\sim b\sim cd\sim eij\sim k\sim mf$
11. $a\sim b\sim c\sim g\sim h\sim i\sim j\sim l + a\sim b\sim c\sim g\sim h\sim ij\sim k + a\sim b\sim c\sim g\sim h\sim i\sim jm + a\sim b\sim c\sim d\sim e\sim j\sim l +$
 $a\sim b\sim c\sim d\sim e\sim jm + a\sim b\sim c\sim d\sim ej\sim k + a\sim b\sim c\sim j\sim l\sim f + a\sim b\sim cj\sim k\sim f + a\sim b\sim c\sim jm\sim f$
12. Not included due to a missing right parenthesis
13. $a + b + c + \sim def\sim g\sim h + ij\sim l + ik\sim l$
14. $ae\sim h + ad\sim h + ace + acd + be + bf$
15. $bei + bdi + bci + aei + aeg + adi + adg + aci + ach + acg + af$
16. $c\sim g\sim i\sim k\sim m + cg\sim h\sim l\sim m + c\sim g\sim hi\sim m + cgi\sim l\sim m + cgi\sim k\sim m + c\sim h\sim k\sim m + b\sim g\sim i\sim k +$
 $a\sim g\sim i\sim k + b\sim g\sim hi + bg\sim h\sim l + a\sim g\sim hi + ag\sim h\sim l + bgi\sim k + bgi\sim l + agi\sim k + agi\sim l +$
 $a\sim h\sim k + b\sim h\sim k + \sim i\sim kf + \sim hif + gf + a\sim e + a\sim d$
17. $acegij + acehik + bdegij + bdehik + acef + bdef$

$$18. ace\sim j\sim k + ace\sim h\sim j + ace\sim g\sim k + bde\sim j\sim k + bde\sim h\sim j + bde\sim g\sim k + bde\sim i + ace\sim i$$

$$19. aceh\sim f + bdeh\sim f + acegf + bdegf$$

$$20. \sim a\sim bd\sim e\sim gf + \sim abc\sim e\sim gf$$

Appendix C General Form TCAS Predicates and Fault Examples

Reprinted [46]

Predicate 1: $\sim(ab)(d\sim e\sim f + \sim de\sim f + \sim d\sim e\sim f)(ac(d + e)h + a(d + e)\sim h + b(e + f))$

Predicate 4: $a(\sim b + \sim c)d + e$

Predicate 6: $(\sim ab + a\sim b)\sim(cd)(f\sim g\sim h + \sim fg\sim h + \sim f\sim g\sim h)\sim(jk)((ac + bd)e(f + (i(gj + hk))))$

Predicate 8: $(\sim ab + a\sim b)\sim(cd)\sim(gh)((ac + bd)e(fg + \sim fh))$

Predicate 9: $\sim(cd)(\sim ef\sim g\sim a(bc + \sim bd))$

Predicate 10: $a\sim b\sim cd\sim ef(g + \sim g(h + i))\sim(jk + \sim jl + m)$

Predicate 13: $a + b + c + \sim c\sim def\sim g\sim h + i(j + k)\sim l$

Predicate 14: $ac(d + e)h + a(d + e)\sim h + b(e + f)$

Predicate 19: $(ac + bd)e(fg + \sim fh)$

Predicate 20: $\sim ef\sim g\sim a(bc + \sim bd)$

Appendix D Minimal DNF, Minimal CNF and MUMCUT Extension Test Sets

(From section 5.1)

This appendix gives details on minimal DNF, minimal CNF, and MUMCUT extension test sets for four TCAS predicates.

PREDICATE 4

Minimal DNF: $a \sim bd + a \sim cd + e$

MUTP test set (4 tests):

10110 term $a \sim bd$
11010 term $a \sim cd$
00001 term e
11111 term e

The MUTP criterion is infeasible for terms $a \sim bd$ and $a \sim cd$, so PCUTPNFP tests are needed for the literals in these terms. Since term e is a single-literal term, any NFP for any other literal will also be an NFP for literal e .

Additional tests needed for a PCUTPNFP test set (5 tests):

00110 term $a \sim bd$, literal a
11110 term $a \sim bd$, literal b and term $a \sim cd$, literal c
10100 term $a \sim bd$, literal d
01010 term $a \sim cd$, literal a
11000 term $a \sim cd$, literal d

Since the PCUTPNFP criterion is feasible for every literal, MNFP tests are not required.

Minimal CNF: $(a + e)(\sim b + \sim c + e)(d + e)$

MUFP test set (5 tests):

00110 term $a + e$
01010 term $a + e$
11110 term $\sim b + \sim c + e$
10100 term $d + e$
11000 term $d + e$

Since the MUFP criterion is infeasible for each term and no term consists of a single literal, PCUFPNTP tests are needed for each literal.

Additional tests needed for a PCUFPNTP test set (5 tests):

10110 term $a + e$, literal a and term $\sim b + \sim c + e$, literal b and term $d + e$, literal d
 00111 term $a + e$, literal e
 11010 term $\sim b + \sim c + e$, literal c
 11111 term $\sim b + \sim c + e$, literal e
 10101 term $d + e$, literal e

Since PCUFNPNT is feasible for every literal, MNTP tests are not required.

MUMCUT extension: $a \sim bd + a \sim cd + e$

1. All UTP test set (15 tests):

10110 term $a \sim bd$
 11010 term $a \sim cd$
 XXXX1 term e

where XXXX is any combination of values except 1001, 1011, or 1101

2. OTP test set (1 test):

10011 terms $a \sim bd$ and $a \sim cd$, terms $a \sim bd$ and e , terms $a \sim cd$ and e

3. All NFP test set (13 tests):

XXXX0 all literals
 where XXXX is any combination of values except 1001, 1011, or 1101

Any point where e is FALSE and terms $a \sim bd$ and terms $a \sim cd$ are both FALSE is an NFP for literal e . The NFPs for the other literals are subsets of the set of all NFPs for literal e .

4. n-MNFP test set

For $n > 1$, no n-NFPs exist as every FALSE point is a 1-NFP for literal e since literal e is in a single-literal term.

PREDICATE 9

Minimal DNF: $\sim a \sim b \sim cd \sim ef \sim g + \sim abc \sim d \sim ef \sim g$

MUTP test set (2 tests):

0001010 term $\sim a \sim b \sim cd \sim ef \sim g$

0110010 term $\sim abc \sim d \sim ef \sim g$

The MUTP criterion is feasible for both terms (as each term contains all unique literals there are no external literals to vary). Thus, only a single NFP is needed for each literal as follows:

NFP test set (14 tests):

1001010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal a

0101010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal b

0011010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal c

0000010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal d

0001110 term $\sim a \sim b \sim cd \sim ef \sim g$, literal e

0001000 term $\sim a \sim b \sim cd \sim ef \sim g$, literal f

0001011 term $\sim a \sim b \sim cd \sim ef \sim g$, literal g

1110010 term $\sim abc \sim d \sim ef \sim g$, literal a

0010010 term $\sim abc \sim d \sim ef \sim g$, literal b

0100010 term $\sim abc \sim d \sim ef \sim g$, literal c

0111010 term $\sim abc \sim d \sim ef \sim g$, literal d

0110110 term $\sim abc \sim d \sim ef \sim g$, literal e

0110000 term $\sim abc \sim d \sim ef \sim g$, literal f

0110011 term $\sim abc \sim d \sim ef \sim g$, literal g

Minimal CNF: $\sim a(\sim b + c)(\sim b + \sim d)(b + \sim c)(b + d)\sim ef \sim g$

MUFP test set (12 tests):

1001010 term $\sim a$

1110010 term $\sim a$

0100010 term $\sim b + c$

0111010 term $\sim b + \sim d$

0011010 term $b + \sim c$

0000010 term $b + d$

0001110 term $\sim e$

0110110 term $\sim e$

0001000 term f

0110000 term f

0001011 term g

0110011 term g

Since terms $\sim a$, $\sim e$, f and $\sim g$ are single-literal terms, any NTP for any other literal will also be an NTP for these four literals. Since MUFP is infeasible for terms $(\sim b + c)$ and $(\sim b + \sim d)$ and $(b + \sim c)$ and $(b + d)$, Partial-CUFPNTP tests are needed for each literal in these terms.

Additional tests needed for a Partial-CUFPNTP test set (2 tests):

No test term $\sim b + c$, literal b because in an NTP for literal b in term $\sim b + c$, literal d must be 1

0110010 term $\sim b + c$, literal c and term $\sim b + \sim d$, literal d
 No test term $\sim b + \sim d$, literal b because in an NTP for literal b in term $\sim b + \sim d$, literal c must be 0
 No test term $b + \sim c$, literal b because in an NTP for literal b in term $b + \sim c$, literal d must be 0
 0001010 term $b + \sim c$, literal c and term $b + d$, literal d
 No test term $b + d$, literal b because in an NTP for literal b in term $\sim b + d$, literal c must be 1

Since Partial-CUFPNTP is not feasible for all literals, MNTP tests are required as specified below.
 However, each of these tests overlaps with the two tests to satisfy Partial-CUFPNTP given above, so no additional tests are required.

0001010 term $\sim b + c$, literal b
 0001010 term $\sim b + \sim d$, literal b
 0110010 term $b + \sim c$, literal b
 0110010 term $b + d$, literal b

MUMCUT extension: $\sim a \sim b \sim cd \sim ef \sim g + \sim abc \sim d \sim ef \sim g$

1. All UTP test set (2 tests):

0001010 term $\sim a \sim b \sim cd \sim ef \sim g$
 0110010 term $\sim abc \sim d \sim ef \sim g$

2. OTP test set (0 tests):

Since each term contains all unique literals, no OTPs exist.

3. All NFP test set (14 tests):

1001010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal a
 0101010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal b
 0011010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal c
 0000010 term $\sim a \sim b \sim cd \sim ef \sim g$, literal d
 0001110 term $\sim a \sim b \sim cd \sim ef \sim g$, literal e
 0001000 term $\sim a \sim b \sim cd \sim ef \sim g$, literal f
 0001011 term $\sim a \sim b \sim cd \sim ef \sim g$, literal g
 1110010 term $\sim abc \sim d \sim ef \sim g$, literal a
 0010010 term $\sim abc \sim d \sim ef \sim g$, literal b
 0100010 term $\sim abc \sim d \sim ef \sim g$, literal c
 0111010 term $\sim abc \sim d \sim ef \sim g$, literal d
 0110110 term $\sim abc \sim d \sim ef \sim g$, literal e
 0110000 term $\sim abc \sim d \sim ef \sim g$, literal f
 0110011 term $\sim abc \sim d \sim ef \sim g$, literal g

4. n-MNFP test set (112 tests)

Since every unique literal occurs in each term, test set size consists of all FALSE points except for 1-NFPs.
 128 possible tests exist with 2 TRUE points and 14 1-NFPs, leaving 112 n-NFPs for $n > 1$.

PREDICATE 13

Minimal DNF: $a + b + c + \sim\text{def} \sim g \sim h + ij \sim l + ik \sim l$

MUTP test set (12 tests):

100000000000 term a
100111111111 term a
010000000000 term b
010111111111 term b
001000000000 term c
001111111111 term c
000011000000 term $\sim\text{def} \sim g \sim h$
000011001111 term $\sim\text{def} \sim g \sim h$
000000001100 term $ij \sim l$
000111111100 term $ij \sim l$
000000001010 term $ik \sim l$
000111111010 term $ik \sim l$

Since terms a, b and c are single-literal terms, any NFP for any other literal will also be an NFP for these three literals. No PCUTPNFP tests are needed for term $\sim\text{def} \sim g \sim h$ as the MUTP criterion is feasible for this term. Thus, NFPs for literals in these terms can be chosen to overlap with other NFPs. PCUTPNFP tests are needed for literals in terms $ik \sim l$ and $ij \sim l$. Literal j is the only literal in a multi-literal term that must be fixed (to FALSE) in a UTP for term $ik \sim l$. Literal k is the only literal in a multi-literal term that must be fixed (to FALSE) in a UTP for term $ij \sim l$. Thus, values for literals d, e, f, g and h do not need to be the same in the UTP – NFP pair chosen to satisfy the PCUTPNFP criterion for the literals in terms $ij \sim l$ and $ik \sim l$. This allows the NFPs chosen for literals d, e, f, g and h to overlap with the NFPs chosen for literals i, j, k and l to satisfy the PCUTPNFP criterion.

Additional tests needed for a PCUTPNFP test set (5 tests):

000111000100 term $ij \sim l$, literal i (and an NFP for literal d)
000001001000 term $ij \sim l$, literal j and term $ik \sim l$, literal k (and an NFP for literal e)
000010001101 term $ij \sim l$, literal l (and an NFP for literal f)
000011100010 term $ik \sim l$, literal i (and an NFP for literal g)
000011011011 term $ik \sim l$, literal l (and an NFP for literal h)

Since the PCUTPNFP criterion is feasible, MNFP tests are not required.

Minimal CNF:

$(a + b + c + \sim d + i)(a + b + c + e + i)(a + b + c + f + i)$
 $(a + b + c + \sim g + i)(a + b + c + \sim h + i)(a + b + c + \sim d + \sim l)$
 $(a + b + c + e + \sim l)(a + b + c + f + \sim l)(a + b + c + \sim g + \sim l)$
 $(a + b + c + \sim h + \sim l)(a + b + c + \sim d + j + k)$
 $(a + b + c + e + j + k)(a + b + c + f + j + k)$
 $(a + b + c + \sim g + j + k)(a + b + c + \sim h + j + k)$

MUFP test set (25 tests):

Rather than enumerate all 25 tests, it is noted that each of the first 10 terms contributes 2 UFPs to a MUFP test set as literals j and k can each attain the values 0 and 1 in UFPs for each of these 10 terms. However, the MUFP criterion is still infeasible for each of these terms as there exist other external literals for each

term besides literals j and k which cannot vary in a UFP. The last five terms each have only a single UFP (the value of each external literal cannot vary).

Since the MUFP criterion is infeasible for each term and no term consists of a single literal, PCUFPNTP tests are needed for each literal.

Additional tests needed for a PCUFPNTP test set (63 tests):

Rather than enumerate all 63 tests a counting argument is given. 80 literals appear in the minimal CNF expression and each requires a single NTP that corresponds to one of the UFPs chosen in the MUFP test set. Thus, 80 corresponding NTPs need to be added not accounting for overlapping. However, the following corresponding NTPs overlap:

000011000010 – literals d, e, f, g, h in terms 1-5

000011001011 – literals d, e, f, g, h in terms 6-10

000011001000 – literals d, e, f, g, h in terms 11-15

This reduces the size by $15 - 3 = 12$. Furthermore, the corresponding NTP for literal i overlaps with the corresponding NTP for literal l in terms 1 and 6, terms 2 and 7, terms 3 and 8, terms 4 and 9 and terms 5 and 10. This reduces test set size by another 5 test cases. Thus, test set size is 63.

Since the PCUFPNTP criterion is feasible for every literal, MNTP tests are not required.

MUMCUT extension: $a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{l} + \text{ik}\sim\text{l}$

1. All UTP test set (1377 tests):

Terms a, b and c each have 434 UTPs as follows. A UTP for term a must start with 100, a UTP for term b must start with 010 and a UTP for term c must start with 001. The remaining 9 literals can vary in value for a total of 512 points. However, for 16 of these points, term $\sim\text{def}\sim\text{g}\sim\text{h}$ will be TRUE as 5 of the 9 literals have fixed values to make term $\sim\text{def}\sim\text{g}\sim\text{h}$, leaving the other 4 literals to vary in value. For some of these 16 points terms $\text{ij}\sim\text{l}$ and/or $\text{ik}\sim\text{l}$ will also be TRUE. Of the 9 original remaining literals, 3 must be fixed to make term $\text{ij}\sim\text{l}$ TRUE and the other 6 can vary in value, leaving 64 points. However, for one half of these points literal k will be TRUE so term $\text{ik}\sim\text{l}$ will be TRUE to reduce the size to 32 points. For one of the remaining 32 points, term $\sim\text{def}\sim\text{g}\sim\text{h}$ will be TRUE, leaving a total of 31 points. A similar analysis holds for term $\text{ik}\sim\text{l}$, reducing the original 512 points by an additional 31 points. Thus, test set size for all UTPs for term a is $512 - 16 - 31 - 31 = 434$. The same size occurs for terms b and c.

Term $\sim\text{def}\sim\text{g}\sim\text{h}$ has 13 UTPs. The values of literals a, b and c must all be fixed at 0 in a UTP for term $\sim\text{def}\sim\text{g}\sim\text{h}$. Thus, only the values of literals i, j, k and l can vary for a total of 16 points. However, 3 of these 16 points make either term $\text{ij}\sim\text{l}$ or $\text{ik}\sim\text{l}$ TRUE (or make both TRUE), leaving 13 UTPs.

Term $\text{ij}\sim\text{l}$ has 31 UTPs. The values of literals a, b, c and k must all be fixed at 0 in a UTP for term $\text{ij}\sim\text{l}$. Thus, only the values of literals d, e, f, g and h can vary for a total of 32 points. However, 1 of these 32 points makes term $\sim\text{def}\sim\text{g}\sim\text{h}$ TRUE, leaving 31 UTPs.

The $ik \sim l$ has 31 UTPs. The values of literals a, b, c and j must all be fixed at 0 in a UTP for term $ik \sim l$. Thus, only the values of literals d, e, f, g and h can vary for a total of 32 points. However, 1 of these 32 points makes term $\sim def \sim g \sim h$ TRUE, leaving 31 UTPs.

2. OTP test set (1 test):

111011001110 makes every term TRUE so it makes every combination of two individual terms TRUE.

3. All NFP test set (434 tests):

AN NFP for literal a must start with 000 and make every other term FALSE. Thus, test set size is the same as the number of UTPs for term a (434). This is because both a UTP for term a and an NFP for literal a must make every other term FALSE. The all NFP test set for every other literal in the predicate is a subset of the all NFP test set for literal a.

4. n-MNFP test set (62 tests)

Since terms a, b and c each contain a single literal, n-MNFP does not apply to literals a, b, or c for $n > 1$. A combinatorial argument is made to derive test set size for the other literals. Test set size cannot exceed:

$$2 \sum_{i=1}^m \sum_{r=2}^{n_m} n_m / r! (n_m - r)!$$

which results in a test set size of 68. Term $\sim def \sim g \sim h$ has an n-MNFP test set size of 52 as $n_m = 5$. Terms $ij \sim l$ and $ik \sim l$ each have an n-MNFP test set size of 8 as $n_m = 3$. However, there is overlap amongst the n-MNFP points for terms $ij \sim l$ and $ik \sim l$ such that of the combined 16 points, only 10 are distinct. Thus, test set size is $52 + 10 = 62$.

PREDICATE 20

Minimal DNF: $\sim a \sim bd \sim ef \sim g + \sim abc \sim ef \sim g$

MUTP test set (4 tests):

0001010 term $\sim a \sim bd \sim ef \sim g$
0011010 term $\sim a \sim bd \sim ef \sim g$
0110010 term $\sim abc \sim ef \sim g$
0111010 term $\sim abc \sim ef \sim g$

The MUTP criterion is feasible for both terms. Thus, only a single NFP is needed for each literal as follows:

NFP test set (10 tests):

1001010 term $\sim a \sim bd \sim ef \sim g$, literal a
0101010 term $\sim a \sim bd \sim ef \sim g$, literal b and term $\sim abc \sim ef \sim g$, literal c
0010010 term $\sim a \sim bd \sim ef \sim g$, literal d and term $\sim abc \sim ef \sim g$, literal b
0001110 term $\sim a \sim bd \sim ef \sim g$, literal e
0001000 term $\sim a \sim bd \sim ef \sim g$, literal f
0001011 term $\sim a \sim bd \sim ef \sim g$, literal g
1110010 term $\sim abc \sim ef \sim g$, literal a
0110110 term $\sim abc \sim ef \sim g$, literal e
0110000 term $\sim abc \sim ef \sim g$, literal f
0110011 term $\sim abc \sim ef \sim g$, literal g

Minimal CNF: $\sim a(\sim b + c)(b + d) \sim ef \sim g$

MUFP test set (12 tests):

1001010 term $\sim a$
1110010 term $\sim a$
1100010 term $\sim b + c$
1101010 term $\sim b + c$
1000010 term $b + d$
1010010 term $b + d$
0001110 term $\sim e$
0110110 term $\sim e$
0001000 term f
0110000 term f
0001011 term $\sim g$
0110011 term $\sim g$

Since terms $\sim a$, $\sim e$, f and $\sim g$ are single-literal terms, any NTP for any other literal will also be an NTP for these four literals. The MUFP criterion is infeasible for $(\sim b + c)$ and $(b + d)$. However, the external literals that cannot vary in value for either of these terms are all in single-literal terms. Thus, any NTP will suffice for these four literals.

NTP test set (2 tests):

0001010 term $\sim b + c$, literal b and term $b + d$, literal d
0110010 term $\sim b + c$, literal c and term $b + d$ literal b

MUMCUT extension: $\sim a \sim bd \sim ef \sim g + \sim abc \sim ef \sim g$

1. All UTP test set (4 tests):

0001010 term $\sim a \sim bd \sim ef \sim g$
0011010 term $\sim a \sim bd \sim ef \sim g$
0110010 term $\sim abc \sim ef \sim g$
0111010 term $\sim abc \sim ef \sim g$

2. OTP test set (0 tests):

Since one term contains b and the other contains $\sim b$, no OTPs exist.

3. All NFP test set (20 tests):

1001010 term $\sim a \sim bd \sim ef \sim g$, literal a
1011010 term $\sim a \sim bd \sim ef \sim g$, literal a
0101010 term $\sim a \sim bd \sim ef \sim g$, literal b and term $\sim abc \sim ef \sim g$, literal c
0010010 term $\sim a \sim bd \sim ef \sim g$, literal d and term $\sim abc \sim ef \sim g$, literal b
0000010 term $\sim a \sim bd \sim ef \sim g$, literal d
0001110 term $\sim a \sim bd \sim ef \sim g$, literal e
0011110 term $\sim a \sim bd \sim ef \sim g$, literal e
0001000 term $\sim a \sim bd \sim ef \sim g$, literal f
0011000 term $\sim a \sim bd \sim ef \sim g$, literal f
0001011 term $\sim a \sim bd \sim ef \sim g$, literal g
0011011 term $\sim a \sim bd \sim ef \sim g$, literal g
1110010 term $\sim abc \sim ef \sim g$, literal a
1111010 term $\sim abc \sim ef \sim g$, literal a
0100010 term $\sim abc \sim ef \sim g$, literal c
0110110 term $\sim abc \sim ef \sim g$, literal e
0111110 term $\sim abc \sim ef \sim g$, literal e
0110000 term $\sim abc \sim ef \sim g$, literal f
0111000 term $\sim abc \sim ef \sim g$, literal f
0110011 term $\sim abc \sim ef \sim g$, literal g
0111011 term $\sim abc \sim ef \sim g$, literal g

4. n-MNFP test set (104 tests)

A combinatorial argument is made to derive test set size. Test set size cannot exceed:

$$2 \sum_{m=1}^m \sum_{r=2}^{n_m} n_m / r! (n_m - r)!$$

which results in a test set size of 228 as $m = 2$ and $n_m = 6$ for each term. However, after removing TRUE points and NFPs and accounting for overlap amongst n-MNFP points, the test set size is 104. Note that 104 accounts for all FALSE points except for the 1-NFPs.

Appendix E RACC Test Set Size Analysis

(From section 6.1)

This section gives a detailed analysis of RACC test set size.

For 5 unique literals, it is shown below that RACC test set size can be $n + 2$ tests.

RACC test set size of $n+2$ for $n=5$

$abcd + !a!b!ce$

A RACC test set for literal d must include 1111X and 1110X where X is either 0 or 1 (but X must be the same in each point). A RACC test set for literal e must include 000X1 and 000X0 where X is either 0 or 1 (but X must be the same in each point). There are two possible RACC tests for literal a as literal a appears in two different terms:

Term 1 - 1111X (UTP) and 0111X (NFP)
Term 2 - 000X1 (UTP) and 100X1 (NFP)

There are two possible RACC tests for literal b as literal b appears in two different terms:

Term 1 - 1111X (UTP) and 1011X (NFP)
Term 2 - 000X1 (UTP) and 010X1 (NFP)

There are two possible RACC tests for literal c as literal c appears in two different terms:

Term 1 - 1111X (UTP) and 1101X (NFP)
Term 2 - 000X1 (UTP) and 001X1 (NFP)

Note that no overlap exists amongst any of the NFPs for any of the literals. This is also shown in Table 49. Note from Table 46 that the values of a, b, c prevent any overlap amongst NFPs for any of the unique literals. Thus five NFPs are required (one for each unique literal) and two UTPs are required (one for each term as literal d and literal e appear in different terms). Thus, a total of seven tests are required for RACC which is $n + 2$.

Table 49 Values of a, b and c in NFPs

Literal	Values of a, b, c in NFP
a in term 1	011
a in term 2	100
b in term 1	101
b in term 2	010
c in term 1	110
c in term 2	001
d in term 1	111
e in term 2	000

The constraint that prevents any overlap amongst NFPs is that for each term in the predicate at least three of its literals are negated in every other term. This constraint that prevents NFP overlapping is called the **triple negation constraint**. For the predicate above, note that term 1 contains three literals (a, b, c) that are all negated in term 2. If just two literals had this relationship then NFP overlap would be possible. For example, consider $abc + !a!bd$. Note that an NFP for literal a in term 1 is 011X and that an NFP for literal b in term 2 is 01X1 such that 0111 is an overlapping NFP. This is why the author conjectures that maximum RACC test set size for 4 unique literals is $n + 1$. Not until 5 unique literals exist can one term have 3 literals, a second term have each of the 3 literals negated and each term have one literal the other does not.

For 6 unique literals, it is shown below that RACC test set size can be $n + 2$ tests.

RACC test set size of $n+2 = 2(n-2)$ for $n=6$

$abc + !a!bd + a!be + !abf$

Since literals c, d, e and f each appear in a different term, 4 UTPs are needed for RACC. Note also that the NFPs amongst literals c, d, e and f cannot overlap with each other because the NFP for literal c requires $a=1, b=1$ and the NFP for literal d requires $a=0, b=0$ and the NFP for literal e requires $a=1, b=0$ and the NFP for literal f requires $a=0, b=1$. Thus, at this point 8 tests are needed (4 UTPs and 4 NFPs) to satisfy RACC. No additional tests are needed though because the NFPs for literals a and b can overlap with NFPs for literals c, d, e and f (the triple negation constraint is not satisfied). For example, 011XX0 is an NFP for literal a in term 1 and literal f in term 4 and 101X0X is an NFP for literal b in term 1 and literal e in term 3. Therefore, $n+2=8$ tests are needed for RACC.

Note that for $n=6$, $n+2 = 2(n-2)$. Intuitively, $2(n-2)$ tests are needed because all but 2 literals (a and b) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a and b can overlap with other tests.)

For 7 unique literals, it is shown below that RACC test set size can be $n + 2$ tests.

RACC test set size of $n+2 = 2(n-2) - 1$ for $n=7$

$abc + !a!bd + a!be + !abf + g$

The only difference between this example and the one above is the addition of term 5 which has a single literal g. Thus, each of the 8 UTPs and NFPs chosen for RACC from the prior example can simply be augmented by letting $g=0$ in each of them. Then only one additional test is needed to satisfy RACC, namely a UTP for term 5 which is of the form XXXXXX1 such that the values of X make none of the first 4 terms

true. AN NFP for literal g can overlap with an NFP for another literal. For example, 0111100 is an NFP for literal a in term 1, literal f in term 4 and literal g in term 5. Thus, 9 tests are needed.

Note that for $n=7$, $n+2 = 2(n-2) - 1$. Note also that $2(n-2) - 1 = 2(n-3) + 1$. Intuitively, $2(n-3) + 1$ tests are needed because all but 3 literals (a, b and g) require 2 tests (a UTP and an NFP) that cannot overlap with each other and one literal (literal g) requires a UTP that cannot overlap with any other test. (The tests for literals a and b as well as the NFP test for literal g can overlap with other tests).

Before leaving the examination of maximum RACC test set size for 7 unique literals, note that the following predicate has a RACC test set size of $n+1=8$.

$$abcd + !a!b!ce + a!bcf + a!b!cg$$

This is because of the following two reasons:

1) RACC tests for literals d, e, f and g require 2 tests (a UTP and an NFP) that cannot overlap with each other.

2) NFPs for literals a, b and c can overlap with NFPs for literals d, e, f and g since the triple negation constraint does not hold.

The above example will be called upon later because it is important in generating a general formula for maximum RACC test set size. The important thing to note is that the predicate in the example above repeats the literals a, b and c or their negations in each term. This example is referred to as the **step** example later since the number of literals that repeat in each term steps up by 1.

For 8 unique literals, it is shown below that RACC test set size can be $n + 2$ tests.

RACC test set size of $n+2 = 2(n-3)$ for $n=8$

$$abcd + !abce + a!bcf + ab!cg + !a!bch$$

Since literals d, e, f, g and h each appear in a different term, 5 UTPs are needed for RACC. Note also that the NFPs amongst literals d, e, f, g and h cannot overlap with each other because the NFP for literal d requires $a=1, b=1, c=1$ and the NFP for literal e requires $a=0, b=1, c=1$ and the NFP for literal f requires $a=1, b=0, c=1$ and the NFP for literal g requires $a=1, b=1, c=0$ and the NFP for literal h requires $a=0, b=0, c=1$. Thus, at this point 10 tests are needed (5 UTPs and 5 NFPs) to satisfy RACC. No additional tests are needed because the NFPs for literals a, b and c can overlap with NFPs for literals d, e, f and g (the triple negation constraint is not satisfied). For example, 01110XXX is an NFP for literal a in term 1 and literal e in term 2, 1011X0XX is an NFP for literal b in term 1 and literal f in term 3 and 1101XX0X is an NFP for literal c in term 1 and literal g in term 4.. Therefore, $n+2=10$ tests are needed for RACC.

Note that for $n=6$, $n+2 = 2(n-3)$. Intuitively, $2(n-3)$ tests are needed because all but 3 literals (a, b and c) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a, b and c can overlap with other tests.)

RACC test set size of $n+3 = 2(n-3)$ for $n=9$

$$abcd + !abce + a!bcf + ab!cg + !a!bch + !a!b!ci$$

Since literals d, e, f, g, h and i each appear in a different term, 6 UTPs are needed for RACC. Note also that the NFPs amongst literals d, e, f, g, h and i cannot overlap with each other because the NFP for literal d requires $a=1, b=1, c=1$ and the NFP for literal e requires $a=0, b=1, c=1$ and the NFP for literal f requires

$a=1, b=0, c=1$ and the NFP for literal g requires $a=1, b=1, c=0$ and the NFP for literal h requires $a=0, b=0, c=1$ and the NFP for literal i requires $a=0, b=0, c=0$. Thus, at this point 12 tests are needed (6 UTPs and 6 NFPs) to satisfy RACC. No additional tests are needed though because the NFPs for literals a, b and c can overlap with NFPs for literals d, e, f, g and i (the triple negation constraint is not satisfied).

Note that for $n=9$, $n+3 = 2(n-3)$. Intuitively, $2(n-3)$ tests are needed because all but 3 literals (a, b and c) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a, b and c can overlap with other tests.)

RACC test set size of $n+4 = 2(n-3)$ for $n=10$

$abcd + !abce + a!bcf + ab!cg + !a!bch + !a!b!ci + a!b!cj$

Since literals d, e, f, g, h, i and j each appear in a different term, 7 UTPs are needed for RACC. Note also that the NFPs amongst literals d, e, f, g, h, i and j cannot overlap with each other because the NFP for literal d requires $a=1, b=1, c=1$ and the NFP for literal e requires $a=0, b=1, c=1$ and the NFP for literal f requires $a=1, b=0, c=1$ and the NFP for literal g requires $a=1, b=1, c=0$ and the NFP for literal h requires $a=0, b=0, c=1$ and the NFP for literal i requires $a=0, b=0, c=0$ and the NFP for literal j requires $a=1, b=0, c=0$. Thus, at this point 14 tests are needed (7 UTPs and 7 NFPs) to satisfy RACC. No additional tests are needed though because the NFPs for literals a, b and c can overlap with NFPs for literals d, e, f, g, i and j (the triple negation constraint is not satisfied).

Note that for $n=10$, $n+3 = 2(n-3)$. Intuitively, $2(n-3)$ tests are needed because all but 3 literals (a, b and c) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a, b and c can overlap with other tests.)

RACC test set size of $n+5 = 2(n-3)$ for $n=11$

$abcd + !abce + a!bcf + ab!cg + !a!bch + !a!b!ci + a!b!cj + !ab!ck$

Since literals d, e, f, g, h, i, j and k each appear in a different term, 8 UTPs are needed for RACC. Note also that the NFPs amongst literals d, e, f, g, h, i, j and k cannot overlap with each other because the NFP for literal d requires $a=1, b=1, c=1$ and the NFP for literal e requires $a=0, b=1, c=1$ and the NFP for literal f requires $a=1, b=0, c=1$ and the NFP for literal g requires $a=1, b=1, c=0$ and the NFP for literal h requires $a=0, b=0, c=1$ and the NFP for literal i requires $a=0, b=0, c=0$ and the NFP for literal j requires $a=1, b=0, c=0$ and the NFP for literal k requires $a=0, b=1, c=0$. Thus, at this point 16 tests are needed (8 UTPs and 8 NFPs) to satisfy RACC. No additional tests are needed though because the NFPs for literals a, b and c can overlap with NFPs for literals d, e, f, g, i, j and k (the triple negation constraint is not satisfied).

Note that for $n=11$, $n+3 = 2(n-3)$. Intuitively, $2(n-3)$ tests are needed because all but 3 literals (a, b and c) require 2 tests (a UTP and an NFP) that cannot overlap with each other. (The tests for literals a, b and c can overlap with other tests.)

The pattern described previously from $n=7$ to $n=11$ repeats starting with $n=12$. That is for $n=12$, a single term (containing a new unique literal) can be added to the example for $n=11$ and RACC test set size will be $2(n-3) - 1$ just as it was $2(n-2) - 1$ for $n=7$. Then for $n=13$ to $n=20$, instead of each term containing literals a, b and c (or their negations) plus one additional literal, each term will begin with the literals a, b, c and d (or their negations) plus one additional literal. Thus, for $n=13$ to $n=20$ RACC test set size will be $2(n-4)$ just like it was $2(n-3)$ for $n=8$ to $n=11$.

The reason this pattern repeats is that for n literals there are 2^n possible combinations of how the literals can be combined in terms of being negated or not negated. For example, for $n=3$, there are 8 possible combinations:

abc, ab!c, a!bc, !abc, a!b!c, !ab!c, !a!bc, !a!b!c

Each of these 8 possible combinations can be a term in the predicate and then a unique literal can be added to each term to bring the total number of literals to $3 + 8 = 11$. Note that for $n=3$, $3 + 8 = 11$ is the same as $n + 2^n = 11$. Thus, at $n=12$ it is not possible to add another term containing a new 12th unique literal that also contains some combination of literals a, b and c (or their negations) without repeating a prior combination. Repeating a prior combination means that the NFP for the 12th unique literal can overlap with an NFP for some other unique literal. For example, if term abcl was added to the example given for $n=11$, the NFP for literal l and the NFP for literal d could overlap as each term contains the same combination of literals a, b and c. So once $n=12$ occurs, the **step** example for $n=7$ becomes relevant. That is, adding a single term containing a single literal l to the predicate in the example given for $n=11$ will result in a RACC test set size that is 1 greater than stepping up the number of literals that repeat in each term from 3 to 4. To make this more concrete, compare:

abcd + !abce + a!bcf + ab!cg + !a!bch + !a!b!ci + a!b!cj + !ab!ck + l

to

abcde + abc!df + ab!cdg + a!bcdh + !abcdi + ab!c!dj + a!bc!dk + a!b!cdl

RACC test set size for the first predicate is $2(n-3) - 1 = 2(n-4) + 1$ because all but 4 literals require 2 tests (a UTP and NFP) that do not overlap with each other and literal l requires a UTP that does not overlap with any other test.

RACC test set size for the second predicate is $2(n-4)$ because all but 4 literals require 2 tests (a UTP and NFP) that do not overlap with each other.

Note the similarity between these two predicates and the two predicates examined for $n=7$. This shows how the pattern repeats.

For $n=13$ to $n=20$, the pattern is similar to $n=8$ to $n=11$ except that the number of literals that repeats in each term changes from 3 (a, b and c) to 4 (a, b, c and d). The pattern stops at $n=20$ because there are 16 possible combinations of a, b, c and d in terms of the literals being negated or not negated. Thus, there can be 16 such terms each of which contains some combination of a, b, c and d or their negations plus an additional unique literal to bring the total number of literals to $4 + 16 = 20$. Note that for $n=4$, $4 + 16 = 20$ is the same as $n + 2^n = 20$. Thus, when $n=21$ the pattern continues as it did when $n=7$ and when $n=12$. Finally, note that $n=7$, $n=12$ and $n=21$ all have something in common. There exists an integer y such that $n - (y + 2^y) = 1$. For $n=7$, $y=2$ and for $n=12$, $y=3$ and for $n=21$, $y=4$.

Appendix F RACC and RICC Single Minimal DNF Fault Detection Proof

(From section 6.2)

This appendix shows examples of how RACC and RICC miss detecting various minimal DNF faults as part of theorem 3.

The Literal Negation Fault (LNF)

Consider the following specification, $f = xa + xb + ab$ and the corresponding implementation fault, $f' = \bar{x}a + xb + ab$. If f_x represents the conditions under which literal x determines the outcome of f , then $f_x = \bar{b}a + b\bar{a}$, $f_a = \bar{b}x + b\bar{x}$ and $f_b = \bar{a}x + a\bar{x}$. Constructing a test set for each of the literals yields $f_x = \{010, 011\}$, $f_a = \{010, 110\}$ and $f_b = \{011, 001\}$. In each case, obviously RACC will return the expected outputs 0 and 1 since f_i defines the conditions under which literal i determines f . However, if the tests for f' are run, the outputs again are 0 and 1. More specifically, where a triple represents the Boolean value of (abx):

- (010, 011) for f' yields expected outputs 0 and 1.
- (010, 110) for f' yields expected outputs 0 and 1.
- (001, 011) for f' yields expected outputs 0 and 1.

In general, the way to miss the fault is to step around the term containing the LNF. In other words, the LNF can be missed when each literal in the term containing the LNF is found in at least one other term in f . Note that this does not include literal negations. In this example, note that literal b is true for the RACC tests for literal a and for the RACC tests for literal x . This means that the literal x in xb and the literal a in ab each determine the value of f for their respective test cases, stepping around the xa term which contains the literal negation.

The Term Omission Fault (TOF)

Consider the same specification used to demonstrate the LNF: namely, $f = xa + xb + ab$. An implementation fault in this case could be $f' = xa + xb$. Then, a valid RACC test could be: $f_x = \{100, 101\}$, $f_a = \{001, 101\}$ and $f_b = \{011, 001\}$, or more simply, (100, 101, 001, 011). Then:

- (100, 101) for f' yields expected outputs 0 and 1.
- (001, 101) for f' yields expected outputs 0 and 1.
- (001, 011) for f' yields expected outputs 0 and 1.

The expected outputs of f and f' are the same and the TOF remains undetected. Another example where RACC could miss the TOF would be $f = xa + \overline{xa}$. Clearly, literal x and a always determine f and hence any test for each literal would do, as long as the literal of interest changes from 1 to 0. Hence, if $f' = xa$, then tests (11, 10) for literal x would yield the results 1 and 0, whereas the tests (11, 01) for literal a would yield 1 and 0. Again, the expected outputs of f and f' are the same and the TOF remains undetected.

In general, RACC could miss the TOF if and only if all literals in the omitted term p_i also appear elsewhere in the function (possibly also including literal negations). Consider the following formal argument:

Let f be represented in terms of literal x , or $f = \bigcup_{i=1}^n a_i + x \bigcup_{i=1}^m b_i + \overline{x} \bigcup_{i=1}^k c_i$. In order to determine under

what conditions the literal x determines f , simply find the Boolean Derivative, or $f_x = \prod_{i=1}^n \overline{a_i} [\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i]$.

Suppose now that term b_l was omitted during implementation.

Then $f' = \bigcup_{i=1}^n a_i + x \bigcup_{i=2}^m b_i + \overline{x} \bigcup_{i=1}^k c_i$ and $f'_x = \prod_{i=1}^n \overline{a_i} [\bigcup_{i=2}^m b_i \oplus \bigcup_{i=1}^k c_i]$. Hence, as long as the b_l term is not used during

RACC testing for literal x (which means that some other term also contains x or its negation), the TOF would remain undetected. The preceding logic holds for all literals in the b_1 term and likewise for a TOF in any b_i or c_j term.

The Operator Reference Fault (ORF)

First, consider the case where an OR is incorrectly replaced with an AND. Consequently, let the specification be $f = ab + cd + \overline{abcd}$ and an incorrect implementation be $f' = abcd + \overline{abcd}$. Also, $f_a = (\overline{c} + \overline{d})(b \oplus \overline{bcd})$, $f_b = (\overline{c} + \overline{d})(a \oplus \overline{acd})$, $f_c = (\overline{a} + \overline{b})(d \oplus \overline{abd})$ and $f_d = (\overline{a} + \overline{b})(c \oplus \overline{abc})$. Then, test sets for f could be:

- (1000, 0000) for f' yields expected outputs 0 and 1.
- (0100, 0000) for f' yields expected outputs 0 and 1.
- (0010, 0000) for f' yields expected outputs 0 and 1.
- (0001, 0000) for f' yields expected outputs 0 and 1.

The tests give the same outputs for f and f' and hence RACC misses the ORF here. Consider the formal argument below of how the fault remains undetected. Again, let $f = \bigcup_{i=1}^n a_i + x \bigcup_{i=1}^m b_i + \overline{x} \bigcup_{i=1}^k c_i$ and a faulty

implementation be $f' = \bigcup_{i=2}^n a_i + x \left(a_1 b_1 + \bigcup_{i=2}^m b_i \right) + \overline{x} \bigcup_{i=1}^k c_i$. Thus, the Boolean Derivatives

are $f_x = \prod_{i=1}^n \overline{a_i} [\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i]$ and $f'_x = \prod_{i=2}^n \overline{a_i} \left[\left(a_1 b_1 + \bigcup_{i=2}^m b_i \right) \oplus \bigcup_{i=1}^k c_i \right]$, respectively. Therefore, if the tester is able

to keep the a_i values false, the b_i values false and the c_i values true (c_j values false and the b_i values true when the ORF occurs with a c_i term), then the variable of interest will determine f and f' . In other words, if the tester can uphold that rule for all literals that are combined during the replacement of the OR operator, the ORF will remain undetected.

Now consider the case where an AND is incorrectly replaced with an OR. Consequently, let the specification be $f = abcd + \sim a \sim b \sim c \sim d$ and an incorrect implementation be $f' = ab + cd + \sim a \sim b \sim c \sim d$. Then a determines the value of f when $bcd \oplus \sim b \sim c \sim d$ is true, b determines the value of f when $acd \oplus \sim a \sim c \sim d$ is true, c determines the value of f when $abd \oplus \sim a \sim b \sim d$ is true and d determines the value of f when $abc \oplus \sim a \sim b \sim c$ is true. Using the same test sets as above for the ORF when OR was replaced by AND will yield identical outputs for f and f' and hence RACC misses the ORF.

The Literal Omission Fault (LOF)

This fault is extremely easy for RACC to miss; consider $f = xa + xb$ and a corresponding LOF where $f' = a + xb$. Then, $f_a = \bar{b}x$, $f_b = \bar{a}x$ and $f_x = a + b$. Thus, potential test sets for RACC include:

- (010), (011), where f' yields the expected outputs 0 and 1.
- (001), (101), where f' yields the expected outputs 0 and 1.
- (001), (011), where f' yields the expected outputs 0 and 1.

The expected outputs of f and f' are the same and the LOF remains undetected. If the LOF is generated on a literal that also appears in another term outside of the one containing the LOF, then generating a test for that literal where the term containing the LOF is 0 will miss the fault. Note that in this example, the term containing the LOF (xa) is 0 for both RACC test points for the literal that is omitted (x). In no circumstance will a RACC test on any other literal catch the LOF either. Hence, the possibility of missing the LOF is exceedingly likely. Again, consider a formal argument. From before, if

$f = \bigcup_{i=1}^n a_i + x \bigcup_{i=1}^m b_i + \bar{x} \bigcup_{i=1}^k c_i$, then the Boolean Derivative for x is $f_x = \prod_{i=1}^n \bar{a}_i [\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i]$. If literal x omitted

during implementation, then $f' = (b_1 + \bigcup_{i=1}^n a_i) + x \bigcup_{i=2}^m b_i + \bar{x} \bigcup_{i=1}^k c_i$ and the Boolean derivative

becomes $f'_x = (\bar{b}_1) \prod_{i=1}^n \bar{a}_i [\bigcup_{i=2}^m b_i \oplus \bigcup_{i=1}^k c_i]$. As seen, if you have the term containing the LOF (b_1 here) remain

false when constructing RACC tests for literal x , then the Boolean Derivative for f' remains unaffected, as does the result for f . Considering another literal inside the b_1 term, the only way it could reveal the LOF is if the term being omitted from f to f' is the only literal that is holding b_1 0 initially. However, this statement is impossible because then the missing literal also determines f , which is a contradiction that another literal in b_1 determines f .

The Literal Insertion Fault (LIF)

For an example of where RACC misses the LIF, consider $f = a + xb$ and $f' = xa + xb$. Then, $f_x = \bar{a}b$,

$f_a = \bar{x} + \bar{b}$ and $f_b = \bar{a}x$. Consequently, test sets for RACC include:

- (011), (010), where f' yields the expected outputs 1 and 0.
- (101), (001), where f' yields the expected outputs 1 and 0.
- (011), (001), where f' yields the expected outputs 1 and 0.

The expected outputs of f and f' are the same and the LIF remains undetected. The formal argument

proceeds as follows. Again, let $f = \bigcup_{i=1}^n a_i + x \bigcup_{i=1}^m b_i + \bar{x} \bigcup_{i=1}^k c_i$, the Boolean Derivative for x

be $f_x = \prod_{i=1}^n \bar{a}_i [\bigcup_{i=1}^m b_i \oplus \bigcup_{i=1}^k c_i]$, the implementation be $f' = \bigcup_{i=2}^n a_i + x(a_1 + \bigcup_{i=1}^m b_i) + \bar{x} \bigcup_{i=1}^k c_i$ and the Boolean

Derivative for the implementation be $f'_x = \prod_{i=2}^n \bar{a}_i [(a_1 + \bigcup_{i=1}^m b_i) \oplus \bigcup_{i=1}^k c_i]$. Then, the RACC tests for literal x will

also work for f' , since setting $a_1 = 0$ does not affect the outcome for f'_x . Likewise, inserting the literal will have no bearing on the outcome from f to f' .

The Literal Reference Fault (LRF)

Consider the following specification of $f = ab + cd + ad + bc$ and the implementation fault of $f' = ac + cd + ad + bc$. Then, $f_a = \bar{c}b + d\bar{c}$, $f_b = \bar{a}d + c\bar{d}$, $f_c = \bar{a}b + d\bar{a}$ and $f_d = \bar{b}c + a\bar{b}$. Potential RACC tests for each literal and outputs for f' include:

- (0101), (1101), where f' yields the expected outputs of 0 and 1.
- (0010), (0110), where f' yields the expected outputs of 0 and 1.
- (0100), (0110), where f' yields the expected outputs of 0 and 1.
- (0010), (0011), where f' yields the expected outputs of 0 and 1.

The expected outputs of f and f' are the same and the LRF remains undetected. The first criterion for missing the fault is similar to the LOF; the replaced literal in the term containing the LRF must appear in another term in the Boolean Function. In addition, both the literal replacing the original literal and all other literals originally in the term containing the LRF must be in at least one other term in the function. More specifically, in this example a , b and c must appear in at least one other term of the function in order to step around testing the term containing the LRF.

The RICC criterion was established in order to determine under what conditions a literal does not determine the outcome of a Boolean function f and then flip that literal from 1 to 0. The logic is repeated both when $f = 1$ and $f = 0$, if feasible. RICC tests also can fail to detect 7 of the 9 faults in Lau and Yu's fault hierarchy as shown below.

The Literal Negation Fault (LNF)

Consider the following specification $f = a + b + c$ and an incorrect implementation of $f' = \bar{a} + b + c$. Then, a does not determine the value of f when $b + c$ is true, b does not determine the value of f when $a + c$ is true and c does not determine the value of f when $a + b$ is true.

Consequently, potential RICC tests are:

- (011, 111), yielding 1 for f and f' .
- (001, 011), yielding 1 for f and f' .
- (010, 011), yielding 1 for f and f' .

It is infeasible to fulfill RICC tests and have f evaluate to 0 and therefore, only one pair of test cases is generated for each literal of interest. The results are the same for f and f' ; therefore, RICC does not catch the fault in this case.

The Term Omission Fault (TOF)

Consider the following specification $f = x + ab + \bar{b}c$ and an incorrect implementation of $f' = ab + \bar{b}c$. Then, a does not determine the value of f when $x + \sim b$ is true, b does not determine the value of f when $x + ac + \sim a \sim c$ is true, c does not determine the value of f when $x + b$ is true and x does not determine the value of f when $ab + \sim bc$ is true. Corresponding RICC tests for each literal and the values for f and f' are listed in Table 50 below, where an "x" indicates the value may be 0 or 1 as long as it is the same value when the literal of interest is both 0 and 1.

Table 50 RICC Tests (TOF)

Literal tested	x	a	b	c	f	f'
a	0	0	0	0	0	0
a	0	1	0	0	0	0
a	x	0	0	1	1	1
a	x	1	0	1	1	1
x	1	1	1	x	1	1
x	0	1	1	x	1	1
b	x	1	0	1	1	1
b	x	1	1	1	1	1
b	0	0	1	0	0	0
b	0	0	0	0	0	0
c	0	0	1	0	0	0
c	0	0	1	1	0	0
c	x	1	1	0	1	1
c	x	1	1	1	1	1

It is infeasible to fulfill RICC tests and have f evaluate to 0 when x is the literal of interest and hence, there are only two rows for literal x . The results are the same for f and f' ; therefore, RICC does not catch the fault in this case.

The Operator Reference Fault (ORF)

First, consider the case where an OR is incorrectly replaced with an AND. Consequently, let the specification be $f = ab + c + d$ and an incorrect implementation be $f' = ab + cd$. Then, a does not determine the value of f when $c + d + \sim b$ is true, b does not determine the value of f when $c + d + \sim a$ is true, c does not determine the value of f when $ab + d$ is true and d does not determine the value of f when $ab + c$ is true. RICC tests for each of the variables follow, along with the expected outputs for f' .

- (0000, 1000), (0011, 1011), which yields expected outputs of 0 and 1 for f' .
- (0000, 0100), (0011, 0111), which yields expected outputs of 0 and 1 for f' .
- (1110, 1100), which yields expected outputs of 1 for f' .
- (1100, 1101), which yields expected outputs of 1 for f' .

It is infeasible to fulfill RICC tests and have f evaluate to 0 when either c or d is the literal of interest and hence, there is only one pair of tests for literal c and literal d . Since f and f' give the same outputs for the tests, the ORF (where OR is changed to AND) is missed in this case. Note that if the test case pair (0011, 0001) was chosen for literal c above instead of (1110, 1100) or if the test case pair (0011, 0010) was chosen for literal d above instead of (1100, 1101) then the ORF would have been detected. However, these test pairs are not required by RICC, so RICC does not guarantee the detection of the ORF.

Now consider the case when an AND is incorrectly replaced with an OR. Consequently, let the specification be $f = abcd$ and an incorrect implementation be $f' = ab + cd$. Then a does not determine the value of f when $\sim b + \sim c + \sim d$ is true and b does not determine the value of f when $\sim a + \sim c + \sim d$ is true and c does not determine the value of f when $\sim a + \sim b + \sim d$ is true and d does not determine the value of f when $\sim a + \sim b + \sim c$ is true. RICC tests for each of the variables follow, along with the expected outputs for f' .

- (0000, 1000), which yields expected outputs of 0 for f' .
- (0000, 0100), which yields expected outputs of 0 for f' .
- (0000, 0010), which yields expected outputs of 0 for f' .
- (0000, 00001), which yields expected outputs of 0 for f' .

It is infeasible for f to evaluate to 1 for any literal of interest and therefore, only one test pair is given for each literal. Since f and f' give the same outputs for the tests, the ORF (where AND is changed to OR) is missed in this case. Note that if instead of the above test cases, the test cases (0011, 1011) and (0011, 0111) and (1100, 1110) and (1100, 1101) were chosen to satisfy RICC, then the ORF would have been detected. However, since these test cases are not required by RICC, RICC does not guarantee the detection of the ORF.

The Literal Omission Fault (LOF)

Consider the following specification $f = abc + de$ and an incorrect implementation of $f' = ab + de$. Then, a does not determine the value of f when $de + \sim b + \sim c$ is true, b does not determine the value of f when $de + \sim a + \sim c$ is true, c does not determine the value of f when $de + \sim a + \sim b$ is true, d does not determine the value of f when $abc + \sim e$ is true and e does not determine the value of f when $abc +$

$\sim d$ is true. Corresponding RICC tests for each literal and the values for f and f' are listed in Table 51 below, where an “x” indicates the value may be 0 or 1 as long as it is the same value when the literal of interest is both 0 and 1.

Table 51 RICC Tests (LOF)

Literal tested	a	b	c	d	e	f	f'
a	1	0	x	0	x	0	0
a	0	0	x	0	x	0	0
a	1	x	x	1	1	1	1
a	0	x	x	1	1	1	1
b	0	1	x	0	x	0	0
b	0	0	x	0	x	0	0
b	x	1	x	1	1	1	1
b	x	0	x	1	1	1	1
c	0	x	1	0	x	0	0
c	0	x	0	0	x	0	0
c	x	x	1	1	1	1	1
c	x	x	0	1	1	1	1
d	1	1	1	1	x	1	1
d	1	1	1	0	x	1	1
d	0	x	x	1	0	0	0
d	0	x	x	0	0	0	0
e	1	1	1	x	1	1	1
e	1	1	1	x	0	1	1
e	0	x	x	0	1	0	0
e	0	x	x	0	0	0	0

The results are the same for f and f' ; therefore, RICC does not catch the fault in this case.

The Literal Insertion Fault (LIF)

Consider the following specification $f = ab + cd + e$ and an incorrect implementation of $f' = ab + cd + ae$. Then, a does not determine the value of f when $cd + e + \sim b$ is true, b does not determine the value of f when $cd + e + \sim a$ is true, c does not determine the value of f when $ab + e + \sim d$ is true, d does not determine the value of f when $ab + e + \sim c$ is true and e does not determine the value of f when $ab + cd$ is true. Corresponding RICC tests for each literal and the values for f and f' are listed in

Table 52 below, where an “x” indicates the value may be 0 or 1 as long as it is the same value when the literal of interest is both 0 and 1.

Table 52 RICC Tests (LIF)

Literal tested	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>f'</i>
<i>a</i>	0	x	1	1	x	1	1
<i>a</i>	1	x	1	1	x	1	1
<i>a</i>	0	0	0	x	0	0	0
<i>a</i>	1	0	0	x	0	0	0
<i>b</i>	x	0	1	1	x	1	1
<i>b</i>	x	1	1	1	x	1	1
<i>b</i>	0	0	0	x	0	0	0
<i>b</i>	0	1	0	x	0	0	0
<i>c</i>	1	1	0	x	x	1	1
<i>c</i>	1	1	1	x	x	1	1
<i>c</i>	0	x	0	0	0	0	0
<i>c</i>	0	x	1	0	0	0	0
<i>d</i>	1	1	x	0	x	1	1
<i>d</i>	1	1	x	1	x	1	1
<i>d</i>	0	x	0	0	0	0	0
<i>d</i>	0	x	0	1	0	0	0
<i>e</i>	1	1	x	x	0	1	1
<i>e</i>	1	1	x	x	1	1	1

It is infeasible to fulfill RICC tests and have f evaluate to 0 when e is the literal of interest and hence, there are only two rows for literal e . As seen, the results are the same for f and f' ; therefore, RICC does not catch the fault in this case.

The Literal Reference Fault (LRF)

Consider the following specification $f = ab + cd + ad + bc$ and an incorrect implementation of $f' = ac + cd + ad + bc$. Then a does not determine the value of f when $c + \sim b \sim d$ is true and b does not determine the value of f when $d + \sim a \sim c$ is true and c does not determine the value of f when $a + \sim b \sim d$ is true and d does not determine the value of f when $b + \sim a \sim c$ is true. Corresponding RICC tests for each literal and the values of f and f' are listed in Table 53 below. Note that the tests where all literals are 0 and

all literals are 1 are repeated in the table for clarity, but that it would only be necessary to actually use one of each.

Table 53 RICC Tests (LRF)

Literal tested	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>f</i>	<i>f'</i>
<i>a</i>	0	0	0	0	0	0
<i>a</i>	1	0	0	0	0	0
<i>a</i>	0	1	1	x	1	1
<i>a</i>	1	1	1	x	1	1
<i>b</i>	0	0	0	x	0	0
<i>b</i>	0	1	0	x	0	0
<i>b</i>	1	0	x	1	1	1
<i>b</i>	1	1	x	1	1	1
<i>c</i>	0	0	0	0	0	0
<i>c</i>	0	0	1	0	0	0
<i>c</i>	1	x	0	1	1	1
<i>c</i>	1	x	1	1	1	1
<i>d</i>	0	x	0	0	0	0
<i>d</i>	0	x	0	1	0	0
<i>d</i>	1	1	1	0	1	1
<i>d</i>	1	1	1	1	1	1

The results are the same for *f* and *f'*; therefore, RICC test do not catch the fault in this case.

Appendix G RACC Single Minimal DNF Fault Detection Analysis

(From section 6.2)

This appendix gives details as to how RACC tests miss detecting minimal DNF faults using a single predicate as an example.

Consider the following predicate used in the study: $aceh\sim f + bdeh\sim f + acegf + bdegf$. A RACC test set for this predicate can be formed from $n+2=10$ tests (described later) whereas a Minimal-MUMCUT test set consists of 16 tests as found by Kaminski and Ammann [19]. For this predicate there are 212 potential faults:

1 ENF

4 TNFs (4 terms exist)

4 TOFs (4 terms exist)

3 ORF+s (3 OR operators exist)

20 LOFs (20 literals exist)

20 LNFs (20 literals exist)

16 ORF.s (16 AND operators exist)

120 LRFs (because each term has 5 of the 8 literals)*

24 LIFs (because each term has 5 of the 8 literals)**

*For each of the 4 terms, each of the 5 literals in the term can be replaced by each of the 3 missing literals or the negation of each of the 3 missing literals. Thus, the total number of LRFs is $4 \times 5 \times 3 \times 2 = 120$.

**For each of the 4 terms, each of the 3 missing literals or the negation of each of the 3 missing literals can be inserted. Thus, the total number of LIFs is $4 \times 3 \times 2 = 24$.

Of the 212 faults, RACC tests are guaranteed to detect only 5 of them, namely the 1 ENF and the 4 TNFs. This leaves 207 faults which may or may not be detected. However, of these 207 faults, the RACC

test set chosen below only detects 103 of them. Thus, RACC tests fail to detect 104 of these 207 faults and 104 of the 212 total faults.

There are 8 unique literals (a-h) in the predicate. Literals a, c, e, f and h appear in term 1 and literals b, d and g (amongst others) appear in term 4. Thus, a RACC test can be composed by just considering literals a, c, e, f and h in term 1 and literals b, d and g in term 4. A UTP for term 1 is 10101001 and a UTP for term 4 is 01011110. Thus, these two points satisfy RACC for the case where the predicate evaluates to true. The following are corresponding NFPs for each literal in term 1:

a - 00101001

c - 10001001

e - 10100001

f - 10101101

h - 10101000

The following are corresponding NFPs for b, d and g in term 4:

b - 00011110

d - 01001110

g - 01011100

These NFPs satisfy RACC for the case where the predicate evaluates to false. Thus, the 10 test points together satisfy RACC. This RACC test set will detect the following 108 faults:

1 ENF

An ENF is detected by any test point in the test set.

4 TNFs

Each TNF is detected by any NFP in the test set and the TNF for term 1 is also detected by the UTP for term 1 and the TNF for term 4 is also detected by the UTP for term 4.

2 TOFs

The TOF for term 1 is detected by the UTP for term 1 and the TOF for term 4 is detected by the UTP for term 4.

2 ORF+s

Replacing OR with AND between terms 1 and 2 is detected by the UTP for term 1 and replacing OR with AND between terms 3 and 4 is detected by the UTP for term 4.

11 LNFs

Each LNF in term 1 is detected by the corresponding NFP for each literal in term 1 and is also detected by the UTP for term 1. Each LNF in term 4 is detected by the UTP for term 4 and the LNFs for b, d and g in term 4 are also detected by the corresponding NFPs for b, d and g. The LNF for g in term 3 is also detected because the corresponding NFP for f in term 1 in the test set also happens to be an NFP for g in term 3.

9 LOFs

Each LOF in term 1 is detected by the corresponding NFP for each literal in term 1. The LOFs for b, d and g in term 4 are also detected by the corresponding NFPs for b, d and g respectively. The LOF for g in term 3 is also detected because the corresponding NFP for f in term 1 in the test set also happens to be an NFP for g in term 3.

16 ORF.s

Each occurrence of an AND replaced by OR in terms 1 and 4 is detected by both the corresponding NFP for the literal on the left of the AND and the corresponding NFP for the literal on the right of the AND. Since each AND in terms 1 and 4 has at least one operand with a corresponding NFP in the test set, all the ORF.s in terms 1 and 4 are detected. All the ORF.s in terms 2 and 3 also happen to be detected. When an ORF. occurs, a single term is split into two terms. Let us call the two terms X and Y. An ORF. can thus be detected by any false point that makes either term X or term Y true. It just so happens that the

corresponding NFPs chosen in the test set make either term X or term Y true for each possible ORF. in terms 2 and 3. For example, consider term 2 (bdeh~f) and the ORF. that splits the term into bd + eh~f. The corresponding NFP for literal g in term 4 (01011100) is such that b=1 and d=1. Thus, this point can detect this particular ORF. because for this point term bd evaluates to true.

57 LRFs

Every LRF involving replacing a literal in term 1 with some literal or the negation of some literal is detected because each literal in term 1 has a corresponding UTP-NFP pair in the test set. Thus, all $5 \times 3 \times 2 = 30$ LRFs involving a literal in term 1 are detected. (Each of the 5 literals in term 1 can be replaced by each of the 3 external literals or their negations.) Every LRF involving replacing literal b, d, or g in term 4 is detected by the RACC test set because each of these literals in term 4 has a corresponding UTP-NFP pair in the test set. Thus, all $3 \times 3 \times 2 = 18$ LRFs involving literals b, d and g in term 4 are detected. Term 4 also contains literals e and f and although the test set does not include a corresponding NFP for either of these literals in term 4, it does include a UTP for term 4. Thus, $1/2$ of the LRFs involving literals e and f in term 4 are detected, meaning an additional $1/2 \times 2 \times 3 \times 2 = 6$ LRFs involving literals e and f in term 4 are detected. To see why consider the UTP for term 4: 01011110. Note that literals e and g are both true in the UTP. Note also that literals a, c and h are all false in the UTP. Thus, replacing literal e or g in term 4 with literal a, c, or h will result in term 4 (and thus the predicate) changing from true to false. Therefore, the test set detects these 6 LRFs. However, the test set does not detect any of the 6 LRFs where $\sim a$ or $\sim c$ or $\sim h$ replaces literal e or g in term 4 because $\sim a$, $\sim c$ and $\sim h$ are all true in the UTP for term 4 in the test set. Thus, replacing literal e or g in term 4 with either $\sim a$, $\sim c$, or $\sim h$ will not change the value of term 4 or the predicate. In regards to term 3, there is no UTP for term 3 and no corresponding NFP for any literal in term 3 in the test set. Thus, $5 \times 3 \times 2 = 30$ LRFs go undetected at first glance. However, literal g in term 3 does have an NFP in the test set as the corresponding NFP for literal f in term 1 (10101101) is also an NFP for literal g in term 3. Thus, for literals a, c, e and f in term 3 all the LRFs go undetected ($4 \times 3 \times 2 = 24$ LRFs). However, for literal g in term 3, $1/2 \times 1 \times 3 \times 2 = 3$ LRFs are detected by the NFP above. Note that in the NFP above, literal g is false but that all other literals in term 3 are true. Thus, if literal g in term 3 is

replaced by a literal (or the negation of a literal) that evaluates to true, then term 3 will change from false to true for the NFP. Since b and d are false in this NFP and h is true, this means that replacing literal g in term 3 with either $\sim b$ or $\sim d$ or h will be detected.

6 LIFs

AN LIF can only be detected by a UTP. No UTP exists for terms 2 or 3 in the test set, so the only LIFs that can be detected are LIFs involving terms 1 and 4. For terms 1 and 4, 1/2 of the LIFs are detected. This is because to detect all LIFs (assuming no equivalent LIFs exist), at least 2 UTPs are needed. Inserting a literal into a term can make a true term false but cannot make a false term true. Thus, the only way to detect the LIF is to select a UTP where the inserted literal (or the inserted negated literal) is false. Consider the UTP for term 1 in the test set: 10101001. Note that literals b, d and g are all false. Thus, inserting literal b, d, or g into term 1 will be detected as term 1 will change from true to false for the UTP given above. However, inserting $\sim b$, $\sim d$, or $\sim g$ into term 1 will not be detected because $\sim b$, $\sim d$ and $\sim g$ are each true for the UTP given above. Thus, inserting any of these into term 1 will cause term 1 to still be true for the UTP given above. Since term 1 can have 6 LIFs and term 4 can have 6 LIFs and since 1/2 the LIFs are detected, a total of 6 LIFs are detected.

RACC fails to detect 104 faults because these faults require additional UTPs or NFPs detect. Varying the RACC test set to focus on other literals and terms would have no impact on the number of faults detected. The actual faults detected would change, but not the number of them.

Appendix H RACC Tests and RACC Fault Detection

(From section 6.2)

For each predicate, this appendix lists the predicate in Minimal DNF, the RACC tests selected for the predicate and the percentage of faults detected by the RACC tests.

1.

$a\sim bd\sim e\sim h\sim f + a\sim b\sim de\sim h\sim f + a\sim bcd\sim e\sim f + a\sim bc\sim de\sim f + \sim ab\sim de\sim f$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
1001000	UTP for term 1 (covers a, b, d, e, f, h)	
1011001	UTP for term 3 (covers c)	
0001000	Corresponding NFP for a in term 1	
1101000	Corresponding NFP for b in term 1	
1001001	Corresponding NFP for c in term 3	
1000000	Corresponding NFP for d in term 1	NFP for e in term 2
1001100	Corresponding NFP for e in term 1	NFP for d in term 2
1001010	Corresponding NFP for f in term 1	
1001001	Corresponding NFP for h in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	5/5	2/5	3/4	22/24	14/29	9/29	21/68	2/8	79/173	46%

2.

$a\sim bc\sim d\sim e\sim gh\sim i\sim f + a\sim b\sim d\sim e\sim g\sim h\sim if + a\sim b\sim c\sim e\sim g\sim h\sim if + a\sim b\sim c\sim d\sim g\sim h\sim if +$
 $a\sim bc\sim d\sim eg\sim h\sim f + a\sim bc\sim d\sim e\sim hi\sim f + a\sim b\sim cd\sim eg\sim h\sim f + a\sim b\sim cd\sim e\sim hi\sim f + a\sim b\sim c\sim deg\sim h\sim f +$
 $a\sim b\sim c\sim de\sim hi\sim f + \sim abc\sim d\sim e\sim hi\sim f + \sim ab\sim cd\sim e\sim hi\sim f + \sim ab\sim c\sim de\sim hi\sim f$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
101000010	UTP for term 1 (covers all literals)	
001000010	Corresponding NFP for a in term 1	
111000010	Corresponding NFP for b in term 1	
100000010	Corresponding NFP for c in term 1	
101100010	Corresponding NFP for d in term 1	
101010010	Corresponding NFP for e in term 1	
101001010	Corresponding NFP for f in term 1	
101000110	Corresponding NFP for g in term 1	
101000000	Corresponding NFP for h in term 1	
101000011	Corresponding NFP for i in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	13/13	1/13	1/12	74/92	9/105	9/105	0/192	0/15	108/548	20%

3.

$\sim a \sim bc \sim g \sim i \sim k \sim m + \sim a \sim bcg \sim h \sim l \sim m + \sim a \sim bc \sim g \sim hi \sim m + \sim a \sim bcgi \sim l \sim m + \sim a \sim bcgi \sim k \sim m +$
 $\sim a \sim bc \sim h \sim k \sim m + \sim ab \sim c \sim g \sim i \sim k + a \sim b \sim c \sim g \sim i \sim k + \sim a \sim bc \sim i \sim kf + \sim ab \sim c \sim g \sim hi + \sim ab \sim cg \sim h \sim l$
 $+ a \sim b \sim c \sim g \sim hi + a \sim b \sim cg \sim h \sim l + \sim a \sim bc \sim h \sim i \sim f + \sim ab \sim cgi \sim k + \sim ab \sim cgi \sim l + a \sim b \sim cgi \sim k +$
 $a \sim b \sim cgi \sim l + a \sim b \sim c \sim h \sim k + \sim ab \sim c \sim h \sim k + a \sim b \sim cgf + \sim ab \sim cgf + \sim a \sim bcgf + a \sim b \sim c \sim d +$
 $a \sim b \sim c \sim e$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
001110010010	UTP for term 1 (covers a, b, c, g, i, k, m)	
001110100100	UTP for term 2 (covers h, l)	
001111010011	UTP for term 9 (covers f)	
100010111111	UTP for term 24 (covers d)	
100100111111	UTP for term 25 (covers e)	
101110010010	Corresponding NFP for a in term 1	NFP for c in term 8
011110010010	Corresponding NFP for b in term 1	NFP for c in term 7
000110010010	Corresponding NFP for c in term 1	NFP for b in term 7 and a in term 8
100110111111	Corresponding NFP for d in term 24 and e in term 25	NFP for k in term 17, f in term 21, l in term 18
001110010011	Corresponding NFP for f in term 9	
001110110010	Corresponding NFP for g in term 1	NFP for i in term 5, h in term 6, f in term 23
001110110100	Corresponding NFP for h in term 2	NFP for i in term 4, f in term 23
001110011010	Corresponding NFP for i in term 1	NFP for h in term 3, g in term 5, h in term 6
001110010110	Corresponding NFP for k in term 1	
001110100110	Corresponding NFP for l in term 2	NFP for k in term 6, f in term 23
001110010011	Corresponding NFP for m in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	25/25	5/25	6/24	111/121	42/146	26/146	335/1764	32/241	583/2493	23%

4.

$a \sim bd + a \sim cd + e$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
10110	UTP for term 1 (covers a, bd)	
11010	UTP for term 2 (covers c)	
11111	UTP for term 3 (covers e)	
00110	Corresponding NFP for a in term 1	
11110	Corresponding NFP for b in term 1, c in term 2, e in term 3	
10100	Corresponding NFP for a in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	3/3	3/3	2/2	4/4	7/7	5/7	28/32	8/12	61/71	86%

5.

$a \sim g \sim i \sim k + a \sim g \sim h \sim l + a \sim g \sim h \sim i + a \sim g \sim i \sim l + a \sim g \sim i \sim k + a \sim h \sim k + a \sim c + a \sim b + f$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
111001001	UTP for term 1 (covers a, g, i, k)	
111010010	UTP for term 2 (covers h, l)	
110011111	UTP for term 7 (covers c)	
101011111	UTP for term 8 (covers b)	
111111111	UTP for term 9 (covers f)	
011001001	Corresponding NFP for a in term 1	
111011111	Corresponding NFP for b in term 2, c in term 7, f in term 9	NFP for l in term 4, k in term 5
111011001	Corresponding NFP for g in term 1	NFP for i in term 5, h in term 6
111011010	Corresponding NFP for h in term 2	NFP for i in term 4
111001101	Corresponding NFP for i in term 1	NFP for h in term 3, g in term 5, h in term 6
111001011	Corresponding NFP for k in term 1	
111010011	Corresponding NFP for l in term 2	NFP for k in term 6

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	9/9	5/9	5/8	19/19	21/28	17/28	158/308	32/73	267/483	55%

6.

$\sim ab \sim cdeg \sim hij \sim k \sim f + a \sim bc \sim deg \sim hij \sim k \sim f + \sim ab \sim cde \sim g \sim h \sim jf + \sim ab \sim cde \sim g \sim h \sim kf + a \sim bc \sim de \sim g \sim h \sim jf + a \sim bc \sim de \sim g \sim h \sim kf$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
01011010110	UTP for term 1 (covers all literals)	
11011010110	Corresponding NFP for a in term 1	
00011010110	Corresponding NFP for b in term 1	
01111010110	Corresponding NFP for c in term 1	
01001010110	Corresponding NFP for d in term 1	
01010010110	Corresponding NFP for e in term 1	
0101110110	Corresponding NFP for f in term 1	
01011000110	Corresponding NFP for g in term 1	
0101101110	Corresponding NFP for h in term 1	
01011010010	Corresponding NFP for i in term 1	
01011010100	Corresponding NFP for j in term 1	
01011010111	Corresponding NFP for k in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	6/6	1/6	1/5	41/52	11/58	11/58	0/144	0/12	72/342	21%

7.

$\sim ab\sim cde\sim g\sim i\sim j + \sim ab\sim cde\sim h\sim i\sim k + a\sim bc\sim de\sim g\sim i\sim j + a\sim bc\sim de\sim h\sim i\sim k + a\sim bc\sim de\sim g\sim k + a\sim bc\sim de\sim h\sim j + \sim ab\sim cde\sim g\sim k + \sim ab\sim cde\sim h\sim j$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
0101101001	UTP for term 1 (covers a, b, c, d, e, g, i, j)	
0101110010	UTP for term 2 (covers h, k)	
1101101001	Corresponding NFP for a in term 1	
0001101001	Corresponding NFP for b in term 1	
0111101001	Corresponding NFP for c in term 1	
0100101001	Corresponding NFP for d in term 1	
0101001001	Corresponding NFP for e in term 1	
0101111001	Corresponding NFP for g in term 1	NFP for h in term 8
0101111010	Corresponding NFP for h in term 2	NFP for g in term 7
0101101101	Corresponding NFP for i in term 1	NFP for k in term 7, h in term 8
0101101011	Corresponding NFP for j in term 1	NFP for k in term 7
0101110011	Corresponding NFP for k in term 2	NFP for j in term 8

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	8/8	2/8	2/7	39/52	20/60	14/60	68/296	4/32	158/524	30%

8.

$\sim ab\sim cde\sim gh\sim f + a\sim bc\sim de\sim gh\sim f + \sim ab\sim cdeg\sim hf + a\sim bc\sim deg\sim hf$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
01011001	UTP for term 1 (covers all literals)	
11011001	Corresponding NFP for a in term 1	
00011001	Corresponding NFP for b in term 1	
01111001	Corresponding NFP for c in term 1	
01001001	Corresponding NFP for d in term 1	
01010001	Corresponding NFP for e in term 1	
01011101	Corresponding NFP for f in term 1	
01011011	Corresponding NFP for g in term 1	
01011000	Corresponding NFP for h in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	4/4	1/4	1/3	21/28	8/32	8/32	N/A	N/A	45/104	42%

9.

$\sim a\sim b\sim cd\sim e\sim gf + \sim abc\sim d\sim e\sim gf$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
0001010	UTP for term 1 (covers all literals)	
1001010	Corresponding NFP for a in term 1	
0101010	Corresponding NFP for b in term 1	
0011010	Corresponding NFP for c in term 1	

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
0000010	Corresponding NFP for d in term 1	
0001110	Corresponding NFP for e in term 1	
0001000	Corresponding NFP for f in term 1	
0001011	Corresponding NFP for g in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	2/2	1/2	1/1	12/12	7/14	7/14	N/A	N/A	31/46	67%

10.

$a \sim b \sim cd \sim eg \sim j \sim l \sim mf + a \sim b \sim cd \sim eh \sim j \sim l \sim mf + a \sim b \sim cd \sim ei \sim j \sim l \sim mf + a \sim b \sim cd \sim eg \sim j \sim k \sim mf + a \sim b \sim cd \sim eh \sim j \sim k \sim mf + a \sim b \sim cd \sim ej \sim k \sim mf$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
1001011000100	UTP for term 1 (covers a, b, c, d, e, f, g, j, l, m)	
1001010101010	UTP for term 5 (covers h, k)	
1001010011010	UTP for term 6 (covers i)	
0001011000100	Corresponding NFP for a in term 1	
1101011000100	Corresponding NFP for b in term 1	
1011011000100	Corresponding NFP for c in term 1	
1000011000100	Corresponding NFP for d in term 1	
1001111000100	Corresponding NFP for e in term 1	
1001001000100	Corresponding NFP for f in term 1	
1001010000100	Corresponding NFP for g in term 1	NFP for h in term 2, i in term 3
1001010001010	Corresponding NFP for h in term 5 and i in term 6	NFP for g in term 4
1001011001100	Corresponding NFP for j in term 1	NFP for g in term 4
1001010101110	Corresponding NFP for k in term 5	NFP for k in term 4
1001011000110	Corresponding NFP for l in term 1	
1001011000101	Corresponding NFP for m in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	6/6	3/6	3/5	54/54	34/60	17/60	141/360	9/24	268/576	47%

11.

$a \sim b \sim c \sim g \sim h \sim i \sim j \sim l + a \sim b \sim c \sim g \sim h \sim i \sim j \sim k + a \sim b \sim c \sim g \sim h \sim i \sim j \sim m + a \sim b \sim c \sim d \sim e \sim j \sim l + a \sim b \sim c \sim d \sim e \sim j \sim m + a \sim b \sim c \sim d \sim e \sim j \sim k + a \sim b \sim c \sim j \sim l \sim f + a \sim b \sim c \sim j \sim k \sim f + a \sim b \sim c \sim j \sim m \sim f$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
1001110000100	UTP for term 1 (covers a, b, c, g, h, i, j, l)	
1000011110111	UTP for term 5 (covers d, e, m)	
1001101111010	UTP for term 8 (covers k, f)	
0001110000100	Corresponding NFP for a in term 1	
1101110000100	Corresponding NFP for b in term 1	
1011110000100	Corresponding NFP for c in term 1	

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
1001011110111	Corresponding NFP for d in term 5	
1000111110111	Corresponding NFP for e in term 5	
100111111010	Corresponding NFP for f in term 8	
1001111000100	Corresponding NFP for g in term 1	NFP for f in term 7
1001110100100	Corresponding NFP for h in term 1	NFP for f in term 7
1001110010100	Corresponding NFP for i in term 1	NFP for f in term 7
1001110001100	Corresponding NFP for j in term 5	NFP for k in term 2
1001101111110	Corresponding NFP for k in term 8	
1001110000110	Corresponding NFP for l in term 1	NFP for m in term 3
1000011110110	Corresponding NFP for m in term 5	NFP for l in term 4

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	9/9	3/9	5/8	54/54	25/63	17/63	135/744	18/96	267/1047	26%

12. Not included due to a missing right parenthesis

13.

a + b + c + ~def~g~h + ij~l + ik~l

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
100000000000	UTP for term 1 (covers a)	
010000000000	UTP for term 2 (covers b)	
001000000000	UTP for term 3 (covers c)	
000011000000	UTP for term 4 (covers d, e, f, g, h)	
000000001100	UTP for term 5 (covers i, j, l)	
000000001010	UTP for term 6 (covers k)	
000000000000	Corresponding NFP for a, b, c in term 1	
000111000000	Corresponding NFP for d in term 4	
000001000000	Corresponding NFP for e in term 4	
000010000000	Corresponding NFP for f in term 4	
000011100000	Corresponding NFP for g in term 4	
000011010000	Corresponding NFP for h in term 4	
000011000100	Corresponding NFP for i in term 5	
000011001001	Corresponding NFP for j in term 5	
000011001000	Corresponding NFP for k in term 6	
000011001101	Corresponding NFP for l in term 5	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	6/6	6/6	5/5	8/8	14/14	12/14	226/244	58/99	336/397	85%

14.

ae~h + ad~h + ace + acd + be + bf

Test		Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with							
1000110		UTP for term 1 (covers a, e, h)								
1011001		UTP for term 4 (covers c, d)								
1100010		UTP for term 6 (covers b, f)								
0000110		Corresponding NFP for a in term 1	NFP for b in term 5							
1000010		Corresponding NFP for b in term 6, e in term 1	NFP for d in term 2							
1001001		Corresponding NFP for c in term 4	NFP for h in term 2							
1010001		Corresponding NFP for d in term 4	NFP for e in term 3							
1100000		Corresponding NFP for f in term 6	NFP for e in term 5, d in term 2							
1000111		Corresponding NFP for h in term 1	NFP for c in term 3, b in term 5							
ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	6/6	3/6	4/5	10/10	14/16	13/16	98/136	13/40	162/236	69%

15.

bei + bdi + bci + aei + aeg + adi + adg + aci + ach + acg + af

Test		Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with							
010010001		UTP for term 1 (covers b, e, i)								
100100100		UTP for term 7 (covers a, d, g)								
101000010		UTP for term 9 (covers c, h)								
100101000		UTP for term 11 (covers f)								
000100100		Corresponding NFP for a in term 7								
000010001		Corresponding NFP for b in term 1	NFP for a in term 4							
100000010		Corresponding NFP for c in term 9								
100000100		Corresponding NFP for d in term 7	NFP for e in term 5, c in term 10							
010000001		Corresponding NFP for e in term 1	NFP for d in term 2, c in term 3							
100100000		Corresponding NFP for f in term 11, g in term 7	NFP for i in term 6							
101000000		Corresponding NFP for h in term 9	NFP i in term 8, g in term 10							
010010000		Corresponding NFP for i in term 1								
ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	11/11	4/11	6/10	21/21	19/32	17/32	171/388	25/99	275/605	45%

16.

c~g~i~k~m + cg~h~l~m + c~g~hi~m + cgi~l~m + cgi~k~m + c~h~k~m + b~g~i~k + a~g~i~k + b~g~hi + bg~h~l + a~g~hi + ag~h~l + bgi~k + bgi~l + agi~k + agi~l + a~h~k + b~h~k + ~i~kf + ~hif + gf + a~e + a~d

Test		Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with							
001110010010		UTP for term 1 (covers c, g, i, k, m)								
010110100101		UTP for term 10 (covers b, h, l)								
000111110010		UTP for term 21 (covers f)								

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
100100010110	UTP for term 22 (covers a, e)	
100010010110	UTP for term 23 (covers d)	
000100010010	Corresponding NFP for a in term 22	NFP for b in term 7, a in term 8, f in term 19
000110100101	Corresponding NFP for b in term 10	NFP for a in term 12
000110010010	Corresponding NFP for c in term 1	NFP for b in term 7, a in term 8, f in term 19
100110010110	Corresponding NFP for d in term 23 and e in term 22	NFP for k in term 8
000110110010	Corresponding NFP for f in term 21	NFP for f in term 19
001110110010	Corresponding NFP for g in term 1	NFP for i in term 5, h in term 6, f in term 19
010110110101	Corresponding NFP for h in term 10	NFP for i in term 13
001110011010	Corresponding NFP for i in term 1	NFP for h in term 3, g in term 5, h in term 6
001110010110	Corresponding NFP for k in term 1	
010110100111	Corresponding NFP for l in term 10	
001110010011	Corresponding NFP for m in term 1	NFP b in term 7, a in term 8, f in term 19

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	23/23	5/23	6/22	59/64	25/87	22/87	316/1390	45/283	502/1980	25%

17.

acegij + acehik + bdegij + bdehik + acef + bdef

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
10101010110	UTP for term 1 (covers a, c, e, g, i, j)	
01011001101	UTP for term 4 (covers b, d, h, k)	
10101100000	UTP for term 5 (covers f)	
00101010110	Corresponding NFP for a in term 1	
00011001101	Corresponding NFP for b in term 4	
10001010110	Corresponding NFP for c in term 1	
01001001101	Corresponding NFP for d in term 2	
10100010110	Corresponding NFP for e in term 1	
10101000000	Corresponding NFP for f in term 5	
10101000110	Corresponding NFP for g in term 1	
01011000101	Corresponding NFP for h in term 4	NFP for f in term 6
10101010010	Corresponding NFP for i in term 1	
10101010100	Corresponding NFP for j in term 1	
01011001100	Corresponding NFP for k in term 4	NFP for f in term 6

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	6/6	3/6	4/5	26/26	17/32	12/32	154/352	17/64	240/524	46%

18.

ace~j~k + ace~h~j + ace~g~k + bde~j~k + bde~h~j + bde~g~k + bde~i + ace~i

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
1010111100	UTP for term 1 (covers a, c, e, j, k)	
1010110101	UTP for term 2 (covers h)	
0101101110	UTP for term 6 (covers b, d, g)	
0101111011	UTP for term 7 (covers i)	
0010111100	Corresponding NFP for a in term 1	
0001101110	Corresponding NFP for b in term 6	
1000111100	Corresponding NFP for c in term 1	
0100101110	Corresponding NFP for d in term 6	
1010011100	Corresponding NFP for e in term 1	
0101111110	Corresponding NFP for g in term 6	NFP for j in term 4
1010111101	Corresponding NFP for h in term 2	NFP for i in term 8
0101111111	Corresponding NFP for i in term 7	
1010111110	Corresponding NFP for j in term 1	NFP for i in term 8
1010111101	Corresponding NFP for k in term 1	NFP for i in term 8

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	8/8	4/8	5/7	30/30	21/38	12/38	172/396	21/70	274/596	46%

19.

aceh~f + bdeh~f + acegf + bdegf

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
10101001	UTP for term 1 (covers a, c, e, f, h)	
01011110	UTP for term 4 (covers b, d, g)	
00101001	Corresponding NFP for a in term 1	
00011110	Corresponding NFP for b in term 4	
10001001	Corresponding NFP for c in term 1	
01001110	Corresponding NFP for d in term 4	
10100001	Corresponding NFP for e in term 1	
10101101	Corresponding NFP for f in term 1	NFP for g in term 3
01011100	Corresponding NFP for g in term 1	
10101000	Corresponding NFP for h in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	4/4	2/4	2/3	16/16	11/20	9/20	57/120	6/24	108/212	51%

20.

$\sim a \sim b d \sim e \sim g f + \sim a b c \sim e \sim g f$

Test	Description and literal coverage	NFPs for other literals that the Corresponding NFP overlaps with
0001010	UTP for term 1 (covers a, b, d ,e, f, g)	
0111010	UTP for term 4 (covers c)	
1001010	Corresponding NFP for a in term 1	
0101010	Corresponding NFP for b in term 1 and c in term 2	
0000010	Corresponding NFP for d in term 1	
0001110	Corresponding NFP for e in term 1	
0001000	Corresponding NFP for f in term 1	
0001011	Corresponding NFP for g in term 1	

ENF	TNF	TOF	ORF+	ORF.	LNF	LOF	LRF	LIF	Total	Percent
1/1	2/2	2/2	1/1	10/10	12/12	7/12	19/24	2/4	56/68	82%

Appendix I TRF-TIF Logic Mutations

(From section 7.2)

This Appendix gives the TRF-TIF logic mutations for one of the predicates examined. This predicate has 12 unique literals so 4096 tests are possible. However, to satisfy Minimal-MUMCUT only 17 tests are needed and there are also only 17 TRF-TIF logic mutations. Killing the resulting 17 mutants guarantees killing 506 other mutants that thus do not need to be generated. These other mutants correspond to mutants based on the mutation operators in the extended fault hierarchy (Figure 9). In other words, detecting the 17 TRF-TIFs guarantees detecting 506 other faults.

After each mutation, a point is given indicating the values the literals need to be assigned to in order to detect the fault. Due to the large number of literals, 1 is used to represent TRUE and 0 is used to represent FALSE. A description of the point is also given in parentheses.

PREDICATE 13

Minimal DNF: $a + b + c + \sim def \sim g \sim h + ij \sim l + ik \sim l$

TRF/LIF mutations are:

ab + ac + ad + ae + af + ag + ah + ai + aj + ak + al + b + c + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

100000000000 (UTP for term a)

ab + ac + a \sim d + a \sim e + a \sim f + a \sim g + a \sim h + a \sim i + a \sim j + a \sim k + a \sim l + b + c + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

100111111111 (UTP for term a)

a + ba + bc + bd + be + bf + bg + bh + bi + bj + bk + bl + c + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

010000000000 (UTP for term b)

a + ba + bc + b \sim d + b \sim e + b \sim f + b \sim g + b \sim h + b \sim i + b \sim j + b \sim k + b \sim l + c + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

010111111111 (UTP for term b)

a + b + ca + cb + cd + ce + cf + cg + ch + ci + cj + ck + cl + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

001000000000 (UTP for term c)

a + b + ca + cb + c \sim d + c \sim e + c \sim f + c \sim g + c \sim h + c \sim i + c \sim j + c \sim k + c \sim l + $\sim def \sim g \sim h + ij \sim l + ik \sim l$

001111111111 (UTP for term c)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{ha} + \sim\text{def}\sim\text{g}\sim\text{hb} + \sim\text{def}\sim\text{g}\sim\text{hc} + \sim\text{def}\sim\text{g}\sim\text{hi} + \sim\text{def}\sim\text{g}\sim\text{hj} + \sim\text{def}\sim\text{g}\sim\text{hk} + \sim\text{def}\sim\text{g}\sim\text{hl} + \text{ij}\sim\text{l} + \text{ik}\sim\text{l}$

000011000000 (UTP for term $\sim\text{def}\sim\text{g}\sim\text{h}$)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{ha} + \sim\text{def}\sim\text{g}\sim\text{hb} + \sim\text{def}\sim\text{g}\sim\text{hc} + \sim\text{def}\sim\text{g}\sim\text{h}\sim\text{i} + \sim\text{def}\sim\text{g}\sim\text{h}\sim\text{j} + \sim\text{def}\sim\text{g}\sim\text{h}\sim\text{k} + \sim\text{def}\sim\text{g}\sim\text{h}\sim\text{l} + \text{ij}\sim\text{l} + \text{ik}\sim\text{l}$

000011001111 (UTP for term $\sim\text{def}\sim\text{g}\sim\text{h}$)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{la} + \text{ij}\sim\text{lb} + \text{ij}\sim\text{lc} + \text{ij}\sim\text{ld} + \text{ij}\sim\text{le} + \text{ij}\sim\text{lf} + \text{ij}\sim\text{lg} + \text{ij}\sim\text{lh} + \text{ij}\sim\text{lk} + \text{ik}\sim\text{l}$

000000001100 (UTP for term $\text{ij}\sim\text{l}$)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{la} + \text{ij}\sim\text{lb} + \text{ij}\sim\text{lc} + \text{ij}\sim\text{ld} + \text{ij}\sim\text{le} + \text{ij}\sim\text{lf} + \text{ij}\sim\text{lg} + \text{ij}\sim\text{lh} + \text{ij}\sim\text{lk} + \text{ik}\sim\text{l}$

000111111100 (UTP for term $\text{ij}\sim\text{l}$)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{l} + \text{ik}\sim\text{la} + \text{ik}\sim\text{lb} + \text{ik}\sim\text{lc} + \text{ik}\sim\text{ld} + \text{ik}\sim\text{le} + \text{ik}\sim\text{lf} + \text{ik}\sim\text{lg} + \text{ik}\sim\text{lh} + \text{ik}\sim\text{lj}$

000000001010 (UTP for term $\text{ik}\sim\text{l}$)

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{l} + \text{ik}\sim\text{la} + \text{ik}\sim\text{lb} + \text{ik}\sim\text{lc} + \text{ik}\sim\text{ld} + \text{ik}\sim\text{le} + \text{ik}\sim\text{lf} + \text{ik}\sim\text{lg} + \text{ik}\sim\text{lh} + \text{ik}\sim\text{lj}$

000111111010 (UTP for term $\text{ik}\sim\text{l}$)

No TIF/LRF mutations are needed for literals a, b, or c as no equivalent LIF mutants exist when inserting a literal from a multi-literal term into term a, b, or c.

No TIF/LOF mutations are needed for literals a, b, or c since a TIF/LOF would result in a TOF, which is detected by a TRF/LIF.

No TIF/LRF mutations are needed for literals in term $\sim\text{def}\sim\text{g}\sim\text{h}$ since 1) any LRF mutation where a literal or the negation of a literal from term a, b, or c replaces a literal in term $\sim\text{def}\sim\text{g}\sim\text{h}$ results in a TOF or an LOF and 2) no equivalent LIF mutants exist when inserting a literal from term $\text{ij}\sim\text{l}$ or term $\text{ik}\sim\text{l}$ into term $\sim\text{def}\sim\text{g}\sim\text{h}$.

TIF/LOF mutations are needed for the literals in term $\sim\text{def}\sim\text{g}\sim\text{h}$ but these mutations can overlap with the TIF/LRF mutations described below for literals in term $\text{ij}\sim\text{l}$ and literals in term $\text{ik}\sim\text{l}$

No TIF/LRF mutations are needed for term $\text{ij}\sim\text{l}$ or term $\text{ik}\sim\text{l}$ involving a literal in any of the first four terms since 1) any LRF mutation where a literal or the negation of a literal from term a, b, or c replaces a literal in term $\text{ij}\sim\text{l}$ or term $\text{ik}\sim\text{l}$ results in a TOF or an LOF and 2) no equivalent LIF mutants exist when inserting a literal from term $\sim\text{def}\sim\text{g}\sim\text{h}$ into term $\text{ij}\sim\text{l}$ or term $\text{ik}\sim\text{l}$.

However, inserting literal $\sim\text{k}$ into term $\text{ij}\sim\text{l}$ results in an equivalent LIF mutant and inserting literal $\sim\text{j}$ into term $\text{ik}\sim\text{l}$ results in an equivalent LIF mutant, so the following TIF/LRF mutations need to be generated:

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{l} + \text{ik}\sim\text{l} + \sim\text{ij}\sim\text{l}\sim\text{k}\sim\text{a}\sim\text{b}\sim\text{cdef}\sim\text{g}\sim\text{h}$

(000111000100) - NFP literal d and for literal i in term $\text{ij}\sim\text{l}$ for d

$a + b + c + \sim\text{def}\sim\text{g}\sim\text{h} + \text{ij}\sim\text{l} + \text{ik}\sim\text{l} + \text{i}\sim\text{j}\sim\text{l}\sim\text{k}\sim\text{a}\sim\text{b}\sim\text{c}\sim\text{d}\sim\text{ef}\sim\text{g}\sim\text{h}$

(000001001000) - NFP for literal e and for literal j in term $ij\sim l$ and for literal k in term $ik\sim l$

$a + b + c + \sim def\sim g\sim h + ij\sim l + ik\sim l + \mathbf{ijl\sim k\sim a\sim b\sim c\sim de\sim f\sim g\sim h}$

(000010001101) - NFP for literal f and for literal l in term $ij\sim l$

$a + b + c + \sim def\sim g\sim h + ij\sim l + ik\sim l + \mathbf{\sim ik\sim l\sim j\sim a\sim b\sim c\sim defg\sim h}$

(00001110010) - this is an NFP for literal g and for literal i in term $ik\sim l$

$a + b + c + \sim def\sim g\sim h + ij\sim l + ik\sim l + \mathbf{ikl\sim j\sim a\sim b\sim c\sim def\sim gh}$

(000011011011) – this is an NFP for literal h and for literal l in term $ik\sim l$

Appendix J Java Programs and TRF-TIF Mutations

(From introduction to Chapter 9)

This appendix gives the source code and example TRF-TIF mutations for 4 java programs used in a study comparing TRF-TIF mutation with muJava.

Cal.java

```
// Returns the number of days between two dates in the same year
// Preconditions: day1 and day2 are in same year
//               1 <= month1 <= month2 <= 12
//               1 <= day1, day2 <= 31
//               range for year: 1 ... 10000

1 public class Cal
2 {
3     public static void main(String[] args)
4     {
5         int month1 = Integer.parseInt(args[0]);
6         int day1 = Integer.parseInt(args[1]);
7         int month2 = Integer.parseInt(args[2]);
8         int day2 = Integer.parseInt(args[5]);
9         int year = Integer.parseInt(args[6]);
10        System.out.println(run(month1,day1,month2,day2,year));
11    }
12
13    private static int run(int month1, int day1, int month2, int day2, int year)
14    {
15        int numDays;
16
17        if ( ( month2 == month1 ) ) numDays = day2 - day1;
18        else
19        {
20            int daysIn[] = {0,31,0,31,30,31,30,31,31,30,31,30,31};
21
22            int m4 = year % 4;
23            int m100 = year % 100;
24            int m400 = year % 400;
25
26            // startTag m100 == 0=F,m400 != 0=F;m4 != 0=T,m100 == 0=T
27            if ( ( m4 != 0 ) || ( m100 == 0 ) && ( m400 != 0 ) ) daysIn[2] = 28;
28
29            else daysIn[2] = 29;
30
31            numDays = day2 + daysIn[month1] - day1;
32
33            for (int i = month1 + 1; (i <= month2-1); i++)
```

```

34     {
35         numDays = daysIn[i] + numDays;
36     }
37 }
38
39     return numDays;
40 }
41 }

```

Tests

Test	Test Values (Program Input)
1	1, 1, 12, 1, 4
2	1, 1, 12, 1, 100
3	1, 1, 12, 1, 400
4	1, 1, 12, 1, 1
5	1, 1, 1, 2, 1

TRF-TIF Mutations

Code Line	Original Predicate	Mutated Predicate	Mutant Type	Tests that kill mutant
17	month2 == month1	TRUE		1,2,3,4
17	month2 == month1	FALSE		5
27	m4 != 0 m100 == 0 && m400 != 0	m4 != 0 && m100 == 0 m4 != 0 && !(m400 != 0) m100 == 0 && m400 != 0	TRF-LIF	4
27	m4 != 0 m100 == 0 && m400 != 0	m4 != 0 m100 == 0 && m400 != 0 && m4 != 0	TRF-LIF	2
27	m4 != 0 m100 == 0 && m400 != 0	m4 != 0 m100 == 0 && m400 != 0 !(m4 != 0) && !(m100 == 0) && m400 != 0	TIF-LOF	1
27	m4 != 0 m100 == 0 && m400 != 0	m4 != 0 m100 == 0 && m400 != 0 !(m4 != 0) && m100 == 0 && !(m400 != 0)	TIF-LOF	3

Prime.java

// Returns the first X prime numbers where X is the input to the program

```
1 public class Prime
2 {
3     public static void main(String[] args)
4     {
5         run(Integer.parseInt(args[0]));
6     }
7
8     private static void run(int input)
9     {
10        int curPrime;
11        int numPrimes;
12        boolean isPrime;
13        int[] primes = new int[100];
14
15        primes[0] = 2;
16        numPrimes = 1;
17        curPrime = 2;
18
19        while ( (numPrimes < input) )
20        {
21            curPrime++;
22            isPrime = true;
23
24            for (int i=0; (i <= numPrimes-1); i++)
25            {
26                if ( (curPrime % primes[i] == 0) )
27                {
28                    isPrime = false;
29                    break;
30                }
31            }
32            if ( (isPrime) )
33            {
34                primes[numPrimes] = curPrime;
35                numPrimes++;
36            }
37        }
38
39        for (int i=0; (i <= numPrimes-1); i++)
40        {
41            System.out.println("Prime: " + primes[i]);
42        }
43    }
44 }
45 }
```

Tests

Test	Test Value (Program Input)
1	4

TRF-TIF Mutations

Code Line	Original Predicate	Mutated Predicate	Tests that kill mutant
26	curPrime % primes[i] == 0	TRUE	1
26	curPrime % primes[i] == 0	FALSE	1
32	isPrime	TRUE	1
32	isPrime	FALSE	1

TestPat.java

// Tests for whether one string contains another string

```

1 public class TestPat
2 {
3     public static void main(String[] argv)
4     {
5         run(argv);
6     }
7
8     private static void run(String[] argv)
9     {
10        final int MAX = 100;
11        char subject[] = new char[MAX];
12        char pattern[] = new char[MAX];
13
14        if ( (argv.length != 2) )
15        {
16            System.out.println("java TestPat String-Subject String-Pattern");
17            return;
18        }
19        subject = argv[0].toCharArray();
20        pattern = argv[1].toCharArray();
21        TestPat testPat = new TestPat();
22        int n=0;
23
24        if ( ((n = testPat.pat(subject,pattern)) == -1) )
25        {
26            System.out.println("Pattern string is not a substring of the subject string");
27        }
28        else
29        {
30            System.out.println("Pattern string begins at the character " + n);
31        }
32    }
33
34    public int pat(char[] subject, char[] pattern)
35    {
36        final int NOTFOUND = -1;
37        int iSub = 0, rtnIndex = NOTFOUND;
38        boolean isPat = false;
39        int subjectLen = subject.length;
40        int patternLen = pattern.length;
41
42        while ( (isPat == false) && (iSub + patternLen - 1 < subjectLen) )
43        {
44            if ( (subject[iSub] == pattern[0]) )
45            {
46                rtnIndex = iSub;
47                isPat = true;
48            }

```

```

49         for (int iPat = 1; (iPat < patternLen); iPat++)
50         {
51             if ( (subject[iSub + iPat] != pattern[iPat]) )
52             {
53                 rtnIndex = NOTFOUND;
54                 isPat = false;
55                 break;
56             }
57         }
58     }
59     iSub++;
60 }
61 return rtnIndex;
62 }
63 }

```

Tests

Test	Test Values (Program Input)
1	"a"
2	"abc", "bc"
3	"abc", "bd"

TRF-TIF Mutations

Code Line	Original Predicate	Mutated Predicate	Tests that kill mutant
14	argv.length != 2	TRUE	2,3
14	argv.length != 2	FALSE	1
24	n = testPat.pat(subject,pattern) == -1	TRUE	2
24	n = testPat.pat(subject,pattern) == -1	FALSE	3
44	subject[iSub] == pattern[0]	TRUE	2,3
44	subject[iSub] == pattern[0]	FALSE	2,3
51	subject[iSub + iPat] != pattern[iPat]	TRUE	2
51	subject[iSub + iPat] != pattern[iPat]	FALSE	3

TriType.java

// Determines if a triangle is equilateral, isosceles, scalene, or invalid

```

1 public class TriType
2 {
3     public static void main (String[] args)
4     {
5         System.out.println(run(Double.parseDouble(args[0]),Double.parseDouble(args[1]),Double.parseDouble(ar
6             gs[2])));
7     }
8     private static int run(double Side1, double Side2, double Side3)
9     {
10         int triOut;
11
12         if ( (Side1 <= 0) || (Side2 <= 0) || (Side3 <= 0) )
13         {
14             triOut = 4;
15
16             return triOut;
17         }
18
19         triOut = 0;
20
21         if ( (Side1 == Side2) )
22         {
23             triOut = triOut + 1;
24         }
25         if ( (Side1 == Side3) )
26         {
27             triOut = triOut + 2;
28         }
29         if ( (Side2 == Side3) )
30         {
31             triOut = triOut + 3;
32         }
33         if ( (triOut == 0) )
34         {
35             // startTag Side1+Side2 <= Side3=T,Side2+Side3 <= Side1=T;Side1+Side2 <=
36             Side3=T,Side1+Side3 <=
37             // Side2=T;Side2+Side3 <= Side1=T,Side1+Side3 <= Side2=T
38
39             if ( (Side1+Side2 <= Side3) || (Side2+Side3 <= Side1) || (Side1+Side3 <= Side2) )
40             {
41                 triOut = 4;
42             }
43             else
44             {
45                 triOut = 1;
46             }
47         }
48     }
49 }

```

```

46
47         return triOut;
48     }
49
50     if ( (triOut > 3) )
51     {
52         triOut = 3;
53     }
54     // startTag Side1+Side2 <= Side3=T,Side2+Side3 <= Side1=T;Side1+Side2 <=
Side3=T,Side1+Side3 <=
55     //         Side2=T;Side2+Side3 <= Side1=T,Side1+Side3 <= Side2=T
56
57     else if ( (Side1+Side2 <= Side3) || (Side2+Side3 <= Side1) || (Side1+Side3 <= Side2) )
58     {
59         triOut = 4;
60     }
61     else
62     {
63         triOut = 2;
64     }
65
66     return triOut;
67 }
68 }

```

Tests

Test	Test Values (Program Input)
1	0, 1, 1
2	1, 0, 1
3	1, 1, 0
4	1, 1, 1
5	1, 2, 1
6	1, 1, 2
7	1, 2, 3
8	3, 1, 2
9	1, 3, 2
10	3, 4, 5
11	2, 2, 1

TRF-TIF Mutations

Code Line	Original Predicate	Mutated Predicate	Mutant Type	Tests that kill mutant
12	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0)	(Side1 <= 0) && (Side2 <= 0) (Side1 <= 0) && (Side3 <= 0) (Side2 <= 0) (Side3 <= 0)	TRF-LIF	1

Code Line	Original Predicate	Mutated Predicate	Mutant Type	Tests that kill mutant
12	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0)	(Side1 <= 0) (Side2 <= 0) && (Side1 <= 0) (Side2 <= 0) && (Side3 <= 0) (Side3 <= 0)	TRF-LIF	2
12	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0)	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0) && (Side1 <= 0) (Side3 <= 0) && (Side2 <= 0)	TRF-LIF	3
12	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0)	(Side1 <= 0) (Side2 <= 0) (Side3 <= 0) !(Side1 <= 0) && !(Side2 <= 0) && !(Side3 <= 0)	TIF-LOF	4-12
21	Side1 == Side2	TRUE		5,7-11
21	Side1 == Side2	FALSE		4,6
25	Side1 == Side3	TRUE		6-11
25	Side1 == Side3	FALSE		4,5
29	Side2 == Side3	TRUE		5-10
29	Side2 == Side3	FALSE		4,11
33	triOut == 0	TRUE		4,5,6,11
33	triOut == 0	FALSE		7-10
38	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) && (Side2+Side3 <= Side1) (Side1+Side2 <= Side3) && (Side1+Side3 <= Side2) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	TRF-LIF	7
38	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) && (Side1+Side2 <= Side3) (Side2+Side3 <= Side1) && (Side1+Side3 <= Side2) (Side1+Side3 <= Side2)	TRF-LIF	8
38	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2) && (Side1+Side2 <= Side3) (Side1+Side3 <= Side2) && (Side2+Side3 <= Side1)	TRF-LIF	9
38	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2) !(Side1+Side2 <= Side3) && !(Side2+Side3 <= Side1) && !(Side1+Side3 <= Side2)	TIF-LOF	10
50	triOut > 3	TRUE		5,6,11
50	triOut > 3	FALSE		4
57	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) && (Side2+Side3 <= Side1) (Side1+Side2 <= Side3) && (Side1+Side3 <= Side2) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	TRF-LIF	6
57	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) && (Side1+Side2 <= Side3) (Side2+Side3 <= Side1) && (Side1+Side3 <= Side2) (Side1+Side3 <= Side2)	TRF-LIF	11

Code Line	Original Predicate	Mutated Predicate	Mutant Type	Tests that kill mutant
57	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2) && (Side1+Side2 <= Side3) (Side1+Side3 <= Side2) && (Side2+Side3 <= Side1)	TRF-LIF	5
57	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2)	(Side1+Side2 <= Side3) (Side2+Side3 <= Side1) (Side1+Side3 <= Side2) !(Side1+Side2 <= Side3) && !(Side2+Side3 <= Side1) && !(Side1+Side3 <= Side2)	TIF-LOF	4

Appendix K Compiere Queries

(From introduction to chapter 10)

This appendix gives details of the Compiere Queries used to compare TRF-TIF mutation with SQLMutation.

Query 1

View the query appears in: C_Invoice_Candidate_v

Schema:

```
<schema>
  <table name="C_Order">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="DocumentNo" type="number"/>
    <column name="DateOrdered" type="date"/>
    <column name="C_DocType_ID" type="number"/>
    <column name="DocStatus" type="varchar"/>
    <column name="InvoiceRule" type="char"/>
  </table>
  <table name="C_OrderLine">
    <column name="QtyOrdered" type="number"/>
    <column name="QtyInvoiced" type="number"/>
    <column name="PriceActual" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="QtyDelivered" type="number"/>
  </table>
  <table name="C_BPartner">
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_InvoiceSchedule_ID" type="number"/>
  </table>
  <table name="C_InvoiceSchedule">
    <column name="C_InvoiceSchedule_ID" type="number"/>
    <column name="InvoiceFrequency" type="char"/>
    <column name="InvoiceDayCutoff" type="number"/>
    <column name="InvoiceDay" type="number"/>
  </table>
  <table name="C_DocType">
    <column name="C_DocType_ID" type="number"/>
    <column name="DocBaseType" type="varchar"/>
    <column name="DocSubTypeSO" type="varchar"/>
  </table>
</schema>
```

SQL:

```

SELECT o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID, o.C_Order_ID, o.DocumentNo,
o.DateOrdered, o.C_DocType_ID,
SUM((l.QtyOrdered-l.QtyInvoiced)*l.PriceActual) AS TotalLines FROM C_Order o INNER JOIN
C_OrderLine l ON o.C_Order_ID=l.C_Order_ID) INNER JOIN C_BPartner bp ON
(o.C_BPartner_ID=bp.C_BPartner_ID) LEFT OUTER JOIN C_InvoiceSchedule si ON
(bp.C_InvoiceSchedule_ID=si.C_InvoiceSchedule_ID) WHERE o.DocStatus IN ('CO','CL','IP') AND
o.C_DocType_ID IN (SELECT C_DocType_ID FROM C_DocType WHERE DocBaseType='SOO' AND
DocSubTypeSO NOT IN ('ON','OB','WR')) AND l.QtyOrdered <> l.QtyInvoiced AND
(o.InvoiceRule='I' OR o.InvoiceRule='O' OR (o.InvoiceRule='D' AND l.QtyInvoiced<>l.QtyDelivered) OR
(o.InvoiceRule='S' AND bp.C_InvoiceSchedule_ID IS NULL) OR (o.InvoiceRule='S' AND
bp.C_InvoiceSchedule_ID IS NOT NULL AND ( (si.InvoiceFrequency IS NULL OR
si.InvoiceFrequency='D') OR si.InvoiceFrequency='W') OR (si.InvoiceFrequency='T' AND
((o.DateOrdered <= sysdate+si.InvoiceDayCutoff-1 AND sysdate >= o.DateOrdered+si.InvoiceDay-1) OR
(o.DateOrdered <= sysdate+si.InvoiceDayCutoff+14 AND sysdate >= o.DateOrdered+si.InvoiceDay+14))))
OR (si.InvoiceFrequency='M' AND o.DateOrdered <= sysdate+si.InvoiceDayCutoff-1 AND sysdate >=
o.DateOrdered+si.InvoiceDay-1)))) GROUP BY o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID,
o.C_Order_ID, o.DocumentNo, o.DateOrdered, o.C_DocType_ID

```

WHERE clause as a Minimal DNF Predicate:

Letting

```

a=o.DocStatus IN ('CO','CL','IP')
b=o.C_DocType_ID IN (SELECT C_DocType_ID FROM C_DocType WHERE DocBaseType='SOO'
AND DocSubTypeSO NOT IN ('ON','OB','WR'))
c=l.QtyOrdered <> l.QtyInvoiced
d=o.InvoiceRule='I'
e=o.InvoiceRule='O'
f=o.InvoiceRule='D'
g=l.QtyInvoiced<>l.QtyDelivered
h=o.InvoiceRule='S'
i=bp.C_InvoiceSchedule_ID IS NOT NULL
j=si.InvoiceFrequency IS NULL
k=si.InvoiceFrequency='D'
l=si.InvoiceFrequency='W'
m=si.InvoiceFrequency='T'
n=o.DateOrdered <= sysdate+si.InvoiceDayCutoff-1
o=sysdate >= o.DateOrdered+si.InvoiceDay-1
p=o.DateOrdered <= sysdate+si.InvoiceDayCutoff+14
q=sysdate >= o.DateOrdered+si.InvoiceDay+14
r=si.InvoiceFrequency='M'

```

the WHERE clause predicate can be expressed as

$$abc(d + e + fg + hi + h!i(j + k + l + mno + mpq + rno))$$

However the following combinations of literal values are infeasible where a comma separates literals and a semicolon separates combinations and TRUE is represented by 1 and FALSE is represented by 0:

```

d=1,e=1;d=1,f=1;d=1,h=1;e=1,f=1;e=1,h=1;f=1,h=1;j=1,k=1;j=1,l=1;j=1,m=1;j=1,r=1;k=1,l=1;k=1,m=1;k
=1,r=1;l=1,m=1;l=1,r=1;m=1,r=1;i=1,j=0;i=1,k=1;i=1,l=1;i=1,m=1;i=1,n=1;i=1,o=1;i=1,p=1;i=1,q=1;i=1,r
=1;o=0,q=1;n=1,p=0

```

Based on these infeasibilities the literals can be reassigned as follows (notice that si.InvoiceFrequency IS NULL no longer appears in the predicate because it is redundant due to the infeasibilities)

```

a=o.DocStatus IN ('CO','CL','IP')
b=o.C_DocType_ID IN (SELECT C_DocType_ID FROM C_DocType WHERE DocBaseType='SOO'
AND DocSubTypeSO NOT IN ('ON','OB','WR'))
c=l.QtyOrdered <> l.QtyInvoiced
d=o.InvoiceRule='I'
e=o.InvoiceRule='O'
f=o.InvoiceRule='D'
g=l.QtyInvoiced<>l.QtyDelivered
h=o.InvoiceRule='S'
i=bp.C_InvoiceSchedule_ID IS NOT NULL
j=si.InvoiceFrequency='D'
k=si.InvoiceFrequency='W'
l=si.InvoiceFrequency='T'
m=o.DateOrdered <= sysdate+si.InvoiceDayCutoff-1
n=sysdate >= o.DateOrdered+si.InvoiceDay-1
o=o.DateOrdered <= sysdate+si.InvoiceDayCutoff+14
p=sysdate >= o.DateOrdered+si.InvoiceDay+14
q=si.InvoiceFrequency='M'

```

Thus the new infeasible combinations are:

d=1,e=1;d=1,f=1;d=1,h=1;e=1,f=1;e=1,h=1;f=1,h=1;j=1,k=1;j=1,l=1;j=1,q=1;k=1,l=1;k=1,q=1;l=1,q=1;i=1,j=1;i=1,k=1;i=1,l=1;i=1,m=1;i=1,n=1;i=1,o=1;i=1,p=1;i=1,q=1;n=0,p=1;m=1,o=0

The WHERE clause predicate in minimal DNF is:

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn

Mutants generated by the TRF-TIF tool:

27 TRF/LIF mutants are generated as follows:

abcd!e!f!hg + abcd!e!f!hi + abcd!e!f!hj + abcd!e!f!hk + abcd!e!f!hl + abcd!e!f!h!m + abcd!e!f!h!n +
abcd!e!f!h!o + abcd!e!f!h!p + abcd!e!f!hq + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop
+ abchqmn

abcd!e!f!h!g + abcd!e!f!hi + abcd!e!f!hj + abcd!e!f!hk + abcd!e!f!hl + abcd!e!f!h!m + abcd!e!f!h!n +
abcd!e!f!h!o + abcd!e!f!h!p + abcd!e!f!h!q + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop
+ abchqmn

abcd!e!f!h!g + abcd!e!f!hi + abcd!e!f!hj + abcd!e!f!hk + abcd!e!f!h!l + abcd!e!f!h!m + abcd!e!f!h!n +
abcd!e!f!h!o + abcd!e!f!h!p + abcd!e!f!hq + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop
+ abchqmn

abcd!e!f!h!g + abcd!e!f!hi + abcd!e!f!hj + abcd!e!f!h!k + abcd!e!f!hl + abcd!e!f!h!m + abcd!e!f!h!n +
abcd!e!f!h!o + abcd!e!f!h!p + abcd!e!f!hq + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop
+ abchqmn

abcd + abce + abcfg + abchi + abchj + **abchk!d!e!f!j!!q!i!g + abchk!d!e!f!j!!q!i!m +**
abchk!d!e!f!j!!q!i!n + abchk!d!e!f!j!!q!i!o + abchk!d!e!f!j!!q!i!p + abchlmn + abchlop + abchqmn

abcd + abce + abcfg + abchi + abchj + **abchk!d!e!f!j!!q!i!g + abchk!d!e!f!j!!q!i!m + abchk!d!e!f!j!!q!i!n**
+ abchk!d!e!f!j!!q!i!o + abchk!d!e!f!j!!q!i!p + abchlmn + abchlop + abchqmn

abcd + abce + abcfg + abchi + abchj + abchk + **abchlmn!d!e!f!j!k!q!io!pg** + abchlop + abchqmn

abcd + abce + abcfg + abchi + abchj + abchk + **abchlmn!d!e!f!j!k!q!io!p!g** + abchlop + abchqmn

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + **abchlop!d!e!f!j!k!q!in!mg** + abchqmn

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + **abchlop!d!e!f!j!k!q!in!m!g** + abchqmn

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + **abchqmn!d!e!f!j!k!!io!g +**
abchqmn!d!e!f!j!k!!io!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + **abchqmn!d!e!f!j!k!!iog +**
abchqmn!d!e!f!j!k!!iop

44 TIF/LRF mutants are generated as follows:

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!e!f!hbcd!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!e!f!hcd!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!e!f!hd!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!d!e!f!h!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!f!hbce!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!f!hce!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!f!he!g!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!hbcfg!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!hcfg!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!hfg!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!f!d!e!hg!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abcf!g!d!e!h!i!j!k!!m!n!o!p!q

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!j!k!!m!n!o!p!qbchi!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!j!k!!m!n!o!p!qchi!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!j!k!!m!n!o!p!qhi!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!j!k!!m!n!o!p!qi!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abch!i!d!e!f!j!k!!m!n!o!p!q!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!i!k!!qbchj!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!i!k!!qchj!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!i!k!!qhj!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!i!k!!qj!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!i!j!!qbchk!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!i!j!!qchk!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!i!j!!qhk!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!i!j!!qk!g!m!n!o!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!i!j!ko!p!qbchlmn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!i!j!ko!p!qchlmn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!i!j!ko!p!qhlmn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!i!j!ko!p!qlmn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abch!l!d!e!f!i!j!ko!p!qmn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abchl!m!d!e!f!i!j!ko!p!qn!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abchlm!n!d!e!f!i!j!ko!p!q!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!i!j!k!mn!qbchlop!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!i!j!k!mn!qchlop!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!i!j!k!mn!qhlop!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!i!j!k!mn!qllop!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abch!l!d!e!f!i!j!k!mn!qop!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abchl!o!d!e!f!i!j!k!mn!qp!g

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
!a!d!e!f!i!j!k!lobchqmn!g!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
a!b!d!e!f!i!j!k!lochqmn!g!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
ab!c!d!e!f!i!j!k!lohqmn!g!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abc!h!d!e!f!i!j!k!loqmn!g!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abchq!m!d!e!f!i!j!k!lon!g!p

abcd + abce + abcfg + abchi + abchj + abchk + abchlmn + abchlop + abchqmn +
abchqm!n!d!e!f!i!j!k!lo!g!p

Test points needed to kill mutants in terms of literal values:

11110000010011110 detects a TRF-LIF: original true and mutant false
11110010000011111 detects a TRF-LIF: original true and mutant false
11110010000111110 detects a TRF-LIF: original true and mutant false

[illegible]

11100001000011100 detects a TIF-LRF: original false and mutant true
 11100001000101100 detects a TIF-LRF: original false and mutant true
 11100001000110100 detects a TIF-LRF: original false and mutant true
 01100001000101110 detects a TIF-LRF: original false and mutant true
 10100001000101110 detects a TIF-LRF: original false and mutant true
 11000001000101110 detects a TIF-LRF: original false and mutant true
 11100000000101110 detects a TIF-LRF: original false and mutant true
 11100001000001110 detects a TIF-LRF: original false and mutant true
 11100001000101010 detects a TIF-LRF: original false and mutant true
 01100001000011101 detects a TIF-LRF: original false and mutant true
 10100001000011101 detects a TIF-LRF: original false and mutant true
 11000001000011101 detects a TIF-LRF: original false and mutant true
 11100000000011101 detects a TIF-LRF: original false and mutant true
 11100001000001101 detects a TIF-LRF: original false and mutant true
 11100001000010101 detects a TIF-LRF: original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

INSERT INTO C_Order VALUES (1,2,3,4,5,SYSDATE-365,6,'CO','T')
 INSERT INTO C_Order VALUES (14,15,3,4,16,SYSDATE+365,6,'CL','T')
 INSERT INTO C_Order VALUES (24,25,3,4,26,SYSDATE-366,6,'IP','O')
 INSERT INTO C_Order VALUES (27,28,3,4,29,SYSDATE+366,6,'CO','O')
 INSERT INTO C_Order VALUES (30,31,3,4,32,SYSDATE-367,6,'CO','D')
 INSERT INTO C_Order VALUES (37,38,3,4,39,SYSDATE+367,6,'CO','D')
 INSERT INTO C_Order VALUES (40,41,3,4,42,SYSDATE+368,6,'CO','S')
 INSERT INTO C_Order VALUES (43,44,3,4,45,SYSDATE-368,6,'CO','S')
 INSERT INTO C_Order VALUES (49,50,3,4,51,SYSDATE+369,6,'CC','T')
 INSERT INTO C_Order VALUES (55,56,3,4,57,SYSDATE+370,54,'CO','T')
 INSERT INTO C_Order VALUES (61,62,3,4,63,SYSDATE+371,6,'CO','B')
 INSERT INTO C_Order VALUES (64,65,3,4,66,SYSDATE+372,6,'DD','O')
 INSERT INTO C_Order VALUES (68,69,3,4,70,SYSDATE+373,6,'CO','O')
 INSERT INTO C_Order VALUES (71,72,3,4,73,SYSDATE+374,6,'EE','D')
 INSERT INTO C_Order VALUES (75,76,3,4,77,SYSDATE+375,74,'CO','D')
 INSERT INTO C_Order VALUES (82,83,3,4,85,SYSDATE+376,6,'FF','S')
 INSERT INTO C_Order VALUES (87,88,3,4,89,SYSDATE+377,86,'CO','S')
 INSERT INTO C_Order VALUES (90,91,3,4,92,SYSDATE-369,6,'GG','S')
 INSERT INTO C_Order VALUES (94,95,3,4,96,SYSDATE-370,94,'CO','S')
 INSERT INTO C_Order VALUES (97,98,3,4,99,SYSDATE-371,6,'CO','C')
 INSERT INTO C_OrderLine VALUES (7,8,9,10,4,8)
 INSERT INTO C_OrderLine VALUES (17,18,19,20,4,21)
 INSERT INTO C_OrderLine VALUES (58,58,59,60,4,58)
 INSERT INTO C_OrderLine VALUES (78,78,79,80,4,81)
 INSERT INTO C_BPartner VALUES (3,11)
 INSERT INTO C_BPartner VALUES (3,NULL)
 INSERT INTO C_InvoiceSchedule VALUES (11,'D',12,13)
 INSERT INTO C_InvoiceSchedule VALUES (11,'M',22,23)
 INSERT INTO C_InvoiceSchedule VALUES (11,'T',33,34)
 INSERT INTO C_InvoiceSchedule VALUES (11,'W',35,36)
 INSERT INTO C_InvoiceSchedule VALUES (11,'T',46,355)
 INSERT INTO C_InvoiceSchedule VALUES (11,'T',-373,47)
 INSERT INTO C_InvoiceSchedule VALUES (11,'M',48,356)
 INSERT INTO C_InvoiceSchedule VALUES (11,'A',52,53)
 INSERT INTO C_InvoiceSchedule VALUES (11,'E',93,360)
 INSERT INTO C_InvoiceSchedule VALUES (11,'T',-371,362)

```
INSERT INTO C_InvoiceSchedule VALUES (11,'T',98,500)
INSERT INTO C_InvoiceSchedule VALUES (11,'F',-374,100)
INSERT INTO C_InvoiceSchedule VALUES (11,'T',-500,101)
INSERT INTO C_InvoiceSchedule VALUES (11,'M',102,358)
INSERT INTO C_InvoiceSchedule VALUES (11,'M',-375,361)
INSERT INTO C_InvoiceSchedule VALUES (11,'M',104,502)
INSERT INTO C_DocType VALUES ('SOO','AA')
INSERT INTO C_DocType VALUES ('SOO','ON')
INSERT INTO C_DocType VALUES ('XXX','BB')
INSERT INTO C_DocType VALUES ('xxx','OB')
```

Query 2

View the query appears in: C_RfQResponseLine_v

Schema:

```
<schema>
  <table name="C_RfQResponseLineQty">
    <column name="C_RfQResponseLineQty_ID" type="number"/>
    <column name="C_RfQLineQty_ID" type="number"/>
    <column name="Price" type="number"/>
    <column name="Discount" type="number"/>
    <column name="C_RfQResponseLine_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="C_RfQLineQty">
    <column name="C_UOM_ID" type="number"/>
    <column name="BenchmarkPrice" type="number"/>
    <column name="Qty" type="number"/>
    <column name="C_RfQLineQty_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="C_UOM">
    <column name="UOMSymbol" type="varchar"/>
    <column name="C_UOM_ID" type="number"/>
  </table>
  <table name="C_RfQResponseLine">
    <column name="C_RfQResponse_ID" type="number"/>
    <column name="C_RfQResponseLine_ID" type="number"/>
    <column name="C_RfQLine_ID" type="number"/>
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="IsActive" type="char"/>
    <column name="Created" type="date"/>
    <column name="CreatedBy" type="varchar"/>
    <column name="Updated" type="date"/>
    <column name="UpdatedBy" type="varchar"/>
  </table>
  <table name="C_RfQLine">
    <column name="Line" type="number"/>
    <column name="M_AttributeSetInstance_ID" type="number"/>
    <column name="Description" type="varchar"/>
    <column name="Help" type="varchar"/>
    <column name="DateWorkStart" type="date"/>
    <column name="DeliveryDays" type="number"/>
    <column name="C_RfQLine_ID" type="number"/>
    <column name="M_Product_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="M_Product">
    <column name="Name" type="varchar"/>
    <column name="DocumentNote" type="varchar"/>
    <column name="UPC" type="number"/>
    <column name="SKU" type="number"/>
  </table>
```

```

        <column name="Value" type="number"/>
        <column name="M_Product_ID" type="number"/>
    </table>
</schema>

```

SQL:

```

SELECT rrl.C_RfQResponse_ID, rrl.C_RfQResponseLine_ID, rrl.C_RfQLine_ID,
rq.C_RfQResponseLineQty_ID, rq.C_RfQLineQty_ID, rrl.AD_Client_ID, rrl.AD_Org_ID, rrl.IsActive,
rrl.Created, rrl.CreatedBy, rrl.Updated, rrl.UpdatedBy,
'en_US' AS AD_Language, rl.Line, rl.M_Product_ID, rl.M_AttributeSetInstance_ID, COALESCE
(p.Name || rl.M_AttributeSetInstance_ID, rl.Description) AS Name, CASE WHEN p.Name IS NOT NULL
THEN rl.Description END AS Description, p.DocumentNote, p.UPC, p.SKU, p.Value AS ProductValue,
rl.Help, rl.DateWorkStart, rl.DeliveryDays, q.C_UOM_ID, uom.UOMSymbol, q.BenchmarkPrice,
q.Qty, rq.Price, rq.Discount FROM C_RfQResponseLineQty rq
INNER JOIN C_RfQLineQty q ON (rq.C_RfQLineQty_ID=q.C_RfQLineQty_ID) INNER JOIN C_UOM
uom ON (q.C_UOM_ID=uom.C_UOM_ID) INNER JOIN C_RfQResponseLine rrl ON
q.C_RfQResponseLine_ID = rrl.C_RfQResponseLine_ID) INNER JOIN C_RfQLine rl ON
(rl.C_RfQLine_ID=rl.C_RfQLine_ID) LEFT OUTER JOIN M_Product p ON
(rl.M_Product_ID=p.M_Product_ID) WHERE rq.IsActive='Y' AND q.IsActive='Y' AND rrl.IsActive='Y'
AND rl.IsActive='Y'

```

WHERE clause as a Minimal DNF Predicate:

Letting a=rq.IsActive='Y', b=q.IsActive='Y', c=rrl.IsActive='Y', d=rl.IsActive='Y' the WHERE clause predicate in minimal DNF is abcd

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
4 LOF mutants are generated as follows: bcd, acd, abd, abc

Test points needed to kill mutants in terms of literal values:

1111 detects a FALSE mutant:	original true and mutant false
0111 detects an LOF:	original false and mutant true
1011 detects an LOF:	original false and mutant true
1101 detects an LOF:	original false and mutant true
1110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```

INSERT INTO C_RfQResponseLineQty VALUES (6,1,7,8,3,'Y')
INSERT INTO C_RfQResponseLineQty VALUES (9,1,10,11,3,'O')
INSERT INTO C_RfQLineQty VALUES (2,12,13,1,'Y')
INSERT INTO C_RfQLineQty VALUES (2,14,15,1,'P')
INSERT INTO C_UOM VALUES ('A',2)
INSERT INTO C_RfQResponseLine VALUES (16,3,4,17,18,'Y',SYSDATE+1,'B',SYSDATE+2,'C')
INSERT INTO C_RfQResponseLine VALUES (19,3,4,20,21,'N',SYSDATE+3,'D',SYSDATE+4,'E')
INSERT INTO C_RfQResponseLine VALUES (42,43,44,45,46,'T',SYSDATE+7,'U',SYSDATE+8,'V')
INSERT INTO C_RfQLine VALUES (22,23,'F','G',SYSDATE+5,24,4,5,'Y')
INSERT INTO C_RfQLine VALUES (26,27,'H','T',SYSDATE+6,25,4,5,'Q')
INSERT INTO M_Product VALUES ('J','K',30,31,32,5)
INSERT INTO M_Product VALUES ('L','M',33,34,35,5)
INSERT INTO M_Product VALUES ('W','X',47,48,49,50)

```

Query 3

View the query appears in: C_RfQResponseLine_vt

Schema:

```
<schema>
  <table name="C_RfQResponseLineQty">
    <column name="C_RfQResponseLineQty_ID" type="number"/>
    <column name="C_RfQLineQty_ID" type="number"/>
    <column name="Price" type="number"/>
    <column name="Discount" type="number"/>
    <column name="C_RfQResponseLine_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="C_RfQLineQty">
    <column name="C_UOM_ID" type="number"/>
    <column name="BenchmarkPrice" type="number"/>
    <column name="Qty" type="number"/>
    <column name="C_RfQLineQty_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="C_UOM">
    <column name="UOMSymbol" type="varchar"/>
    <column name="C_UOM_ID" type="number"/>
  </table>
  <table name="C_RfQResponseLine">
    <column name="C_RfQResponse_ID" type="number"/>
    <column name="C_RfQResponseLine_ID" type="number"/>
    <column name="C_RfQLine_ID" type="number"/>
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="IsActive" type="char"/>
    <column name="Created" type="date"/>
    <column name="CreatedBy" type="varchar"/>
    <column name="Updated" type="date"/>
    <column name="UpdatedBy" type="varchar"/>
  </table>
  <table name="C_RfQLine">
    <column name="Line" type="number"/>
    <column name="M_Produce_ID" type="number"/>
    <column name="M_AttributeSetInstance_ID" type="number"/>
    <column name="Description" type="varchar"/>
    <column name="Help" type="varchar"/>
    <column name="DateWorkStart" type="date"/>
    <column name="DeliveryDays" type="number"/>
    <column name="C_RfQLine_ID" type="number"/>
    <column name="M_Product_ID" type="number"/>
    <column name="IsActive" type="char"/>
  </table>
  <table name="M_Product">
    <column name="Name" type="varchar"/>
    <column name="DocumentNote" type="varchar"/>
    <column name="UPC" type="number"/>
  </table>
```

```

    <column name="SKU" type="number"/>
    <column name="Value" type="number"/>
    <column name="M_Product_ID" type="number"/>
  </table>
  <table name="AD_Language">
    <column name="AD_Language" type="varchar"/>
    <column name="IsSystemLanguage" type="char"/>
  </table>
</schema>

```

SQL:

```

SELECT rrl.C_RfQResponse_ID, rrl.C_RfQResponseLine_ID, rrl.C_RfQLine_ID,
rq.C_RfQResponseLineQty_ID, rq.C_RfQLineQty_ID, rrl.AD_Client_ID, rrl.AD_Org_ID, rrl.IsActive,
rrl.Created, rrl.CreatedBy, rrl.Updated, rrl.UpdatedBy,
l.AD_Language, rl.Line, rl.M_Product_ID, rl.M_AttributeSetInstance_ID,
COALESCE(p.Name||rl.M_AttributeSetInstance_ID, rl.Description) AS Name, CASE WHEN p.Name IS
NOT NULL THEN rl.Description END AS Description, p.DocumentNote, p.UPC, p.SKU, p.Value AS
ProductValue, rl.Help, rl.DateWorkStart, rl.DeliveryDays, q.C_UOM_ID, uom.UOMSymbol, q.Qty,
rq.Price, rq.Discount FROM C_RfQResponseLineQty rq INNER JOIN C_RfQLineQty q ON
(rq.C_RfQLineQty_ID = q.C_RfQLineQty_ID) INNER JOIN C_UOM uom ON
(q.C_UOM_ID=uom.C_UOM_ID) INNER JOIN C_RfQResponseLine rrl ON
(rq.C_RfQResponseLine_ID=rrl.C_RfQResponseLine_ID) INNER JOIN C_RfQLine rl ON
rrl.C_RfQLine_ID = rl.C_RfQLine_ID) LEFT OUTER JOIN M_Product p ON
(rl.M_Product_ID=p.M_Product_ID) INNER JOIN AD_Language l ON (l.IsSystemLanguage='Y')
WHERE rq.IsActive='Y' AND q.IsActive='Y' AND rrl.IsActive='Y' AND rl.IsActive='Y'

```

WHERE clause as a Minimal DNF Predicate:

Letting a=rq.IsActive='Y', b=q.IsActive='Y', c=rrl.IsActive='Y', d=rl.IsActive='Y' the WHERE clause predicate in minimal DNF is abcd

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
4 LOF mutants are generated as follows: bcd, acd, abd, abc

Test points needed to kill mutants in terms of literal values:

1111 detects a FALSE mutant:	original true and mutant false
0111 detects an LOF:	original false and mutant true
1011 detects an LOF:	original false and mutant true
1101 detects an LOF:	original false and mutant true
1110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```

INSERT INTO C_RfQResponseLineQty VALUES (6,1,7,8,3,'Y')
INSERT INTO C_RfQResponseLineQty VALUES (9,1,10,11,3,'N')
INSERT INTO C_RfQLineQty VALUES (2,12,13,1,'Y')
INSERT INTO C_RfQLineQty VALUES (2,14,15,1,'N')
INSERT INTO C_RfQLineQty VALUES (37,38,39,40,'R')
INSERT INTO C_UOM VALUES ('A',2)
INSERT INTO C_UOM VALUES ('S',41)
INSERT INTO C_RfQResponseLine VALUES (16,3,4,17,18,'Y',SYSDATE+1,'B',SYSDATE+2,'C')
INSERT INTO C_RfQResponseLine VALUES (19,3,4,20,21,'N',SYSDATE+3,'D',SYSDATE+4,'E')
INSERT INTO C_RfQResponseLine VALUES (42,43,44,45,46,'T',SYSDATE+7,'U',SYSDATE+8,'V')
INSERT INTO C_RfQLine VALUES (22,25,23,'F','G',SYSDATE+5,24,4,5,'Y')

```

```
INSERT INTO C_RfQLine VALUES (26,29,27,'H','I',SYSDATE+6,28,4,5,'N')
INSERT INTO M_Product VALUES ('J','K',30,31,32,5)
INSERT INTO M_Product VALUES ('L','M',33,34,35,5)
INSERT INTO M_Product VALUES ('W','X',47,48,49,50)
INSERT INTO AD_Language VALUES ('O','Y')
INSERT INTO AD_Language VALUES ('O','Z')
```

Query 4

View the query appears in: M_InOut_Candidate_v

Schema:

```
<schema>
  <table name="C_Order">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="DocumentNo" type="number"/>
    <column name="DateOrdered" type="date"/>
    <column name="C_DocType_ID" type="number"/>
    <column name="POReference" type="varchar"/>
    <column name="Description" type="varchar"/>
    <column name="SalesRep_ID" type="number"/>
    <column name="DocStatus" type="varchar"/>
    <column name="isDelivered" type="char"/>
    <column name="DeliveryRule" type="char"/>
    <column name="IsDropShip" type="char"/>
  </table>
  <table name="C_OrderLine">
    <column name="M_Warehouse_ID" type="number"/>
    <column name="QtyOrdered" type="number"/>
    <column name="QtyDelivered" type="number"/>
    <column name="PriceActual" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="M_Product_ID" type="number"/>
    <column name="C_Charge_ID" type="number"/>
    <column name="C_OrderLine_ID" type="number"/>
  </table>
  <table name="M_Product">
    <column name="M_Product_ID" type="number"/>
    <column name="IsExcludeAutoDelivery" type="char"/>
  </table>
  <table name="M_InOutLine">
    <column name="M_InOut_ID" type="number"/>
    <column name="C_OrderLine_ID" type="number"/>
  </table>
  <table name="M_InOut">
    <column name="M_InOut_ID" type="number"/>
    <column name="DocStatus" type="varchar"/>
  </table>
  <table name="C_DocType">
    <column name="C_DocType_ID" type="number"/>
    <column name="DocBaseType" type="varchar"/>
    <column name="DocSubTypeSO" type="varchar"/>
  </table>
</schema>
```

SQL:


```

SELECT o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID, o.C_Order_ID, o.DocumentNo,
o.DateOrdered, o.C_DocType_ID,
o.PORreference, o.Description, o.SalesRep_ID, l.M_Warehouse_ID, SUM((l.QtyOrdered-
l.QtyDelivered)*l.PriceActual) AS
TotalLines FROM C_Order o INNER JOIN C_OrderLine l ON (o.C_Order_ID=l.C_Order_ID) WHERE
(o.DocStatus = 'CO' AND o.IsDelivered='N') AND o.C_DocType_ID IN (SELECT C_DocType_ID
FROM C_DocType WHERE DocBaseType = 'SOO' AND DocSubTypeSO NOT IN ('ON','OB','WR'))
AND o.DeliveryRule<>'M' AND (l.M_Product_ID IS NULL OR EXISTS (SELECT * FROM M_Product
p WHERE l.M_Product_ID=p.M_Product_ID AND p.IsExcludeAutoDelivery='N')) AND l.QtyOrdered <>
l.QtyDelivered AND o.IsDropShip='N' AND (l.M_Product_ID IS NOT NULL OR l.C_Charge_ID IS NOT
NULL) AND NOT EXISTS (SELECT * FROM M_InOutLine iol INNER JOIN M_InOut io ON
(iol.M_InOut_ID=io.M_InOut_ID) WHERE iol.C_OrderLine_ID=l.C_OrderLine_ID AND io.DocStatus
IN ('IP','WC'))
GROUP BY o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID, o.C_Order_ID, o.DocumentNo,
o.DateOrdered, o.C_DocType_ID, o.PORreference, o.Description, o.SalesRep_ID, l.M_Warehouse_ID

```

WHERE clause as a Minimal DNF Predicate:

Letting

a=o.DocStatus = 'CO'

b=o.IsDelivered='N'

c=o.C_DocType_ID IN (SELECT C_DocType_ID FROM C_DocType
WHERE DocBaseType='SOO' AND DocSubTypeSO NOT IN ('ON','OB','WR'))

d=o.DeliveryRule<>'M'

e=l.M_Product_ID IS NULL

j=EXISTS (SELECT * FROM M_Product p WHERE l.M_Product_ID=p.M_Product_ID AND
p.IsExcludeAutoDelivery='N'))

f=l.QtyOrdered <> l.QtyDelivered

g=o.IsDropShip='N'

h=l.C_Charge_ID IS NOT NULL

i=NOT EXISTS (SELECT * FROM M_InOutLine iol
INNER JOIN M_InOut io ON (iol.M_InOut_ID=io.M_InOut_ID)
WHERE iol.C_OrderLine_ID=l.C_OrderLine_ID AND io.DocStatus IN ('IP','WC'))

The WHERE clause predicate can be expressed as $abcd(e + j)fg(!e + h)i$

However $e=1, j=1$ is an infeasible combination of literal values and thus the WHERE clause predicate in minimal DNF is

$abcdefghi + abcdjfgi$

Mutants generated by the TRF-TIF tool:

3 TRF/LIF mutants are generated as follows:

abcdefghij + abcdjfgi
abcdefghi + abcdjfgi!e!h
abcdefghi + abcdjfgi!eh

17 TIF/LRF mutants are generated as follows:

abcdefghi + abcdjfgi + !a!jbcdefghi
abcdefghi + abcdjfgi + a!b!jcddefghi
abcdefghi + abcdjfgi + ab!c!jdefghi
abcdefghi + abcdjfgi + abc!d!jefghi

abcdefghi + abcdjfgi + **abcd!e!jfgi**
 abcdefghi + abcdjfgi + **abcde!f!jghi**
 abcdefghi + abcdjfgi + **abcdef!g!jhi**
 abcdefghi + abcdjfgi + **abcdefg!h!ji**
 abcdefghi + abcdjfgi + **abcdefgh!i!j**
 abcdefghi + abcdjfgi + **!a!ebcdjfgi!h**
 abcdefghi + abcdjfgi + **a!b!ecdjfgi!h**
 abcdefghi + abcdjfgi + **ab!c!edjfgi!h**
 abcdefghi + abcdjfgi + **abc!d!ejfgi!h**
 abcdefghi + abcdjfgi + **abcd!j!efgi!h**
 abcdefghi + abcdjfgi + **abcdj!f!egi!h**
 abcdefghi + abcdjfgi + **abcdjf!g!ei!h**
 abcdefghi + abcdjfgi + **abcdjfg!i!e!h**

Test points needed to kill mutants in terms of literal values:

111111110 detects a TRF-LIF: original true and mutant false
 111101111 detects a TRF-LIF: original true and mutant false
 111101101 detects a TRF-LIF: original true and mutant false
 011111110 detects a TIF-LRF: original false and mutant true
 101111110 detects a TIF-LRF: original false and mutant true
 110111110 detects a TIF-LRF: original false and mutant true
 111011110 detects a TIF-LRF: original false and mutant true
 111101110 detects a TIF-LRF: original false and mutant true
 111110110 detects a TIF-LRF: original false and mutant true
 111111010 detects a TIF-LRF: original false and mutant true
 111111100 detects a TIF-LRF: original false and mutant true
 011101101 detects a TIF-LRF: original false and mutant true
 101101101 detects a TIF-LRF: original false and mutant true
 110101101 detects a TIF-LRF: original false and mutant true
 111001101 detects a TIF-LRF: original false and mutant true
 111101101 detects a TIF-LRF: original false and mutant true
 111100101 detects a TIF-LRF: original false and mutant true
 111101001 detects a TIF-LRF: original false and mutant true
 1111011001 detects a TIF-LRF: original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

INSERT INTO C_Order VALUES (1,2,3,4,5,SYSDATE+1,6,'AA','BB',7,'CO','N','A','N')
 INSERT INTO C_Order VALUES (30,31,32,4,33,SYSDATE+2,6,'GG','HH',34,'II','N','D','N')
 INSERT INTO C_Order VALUES (35,36,37,4,38,SYSDATE+3,6,'JJ','KK',39,'CO','E','F','N')
 INSERT INTO C_Order VALUES (41,42,43,4,44,SYSDATE+4,40,'LL','KK',45,'CO','N','G','N')
 INSERT INTO C_Order VALUES (46,47,48,4,49,SYSDATE+5,6,'NN','OO',50,'CO','N','M','N')
 INSERT INTO C_Order VALUES (55,56,57,4,58,SYSDATE+6,6,'PP','QQ',59,'CO','N','T','H')
 INSERT INTO C_OrderLine VALUES (8,9,10,11,4,NULL,12,13)
 INSERT INTO C_OrderLine VALUES (16,17,18,19,4,15,20,13)
 INSERT INTO C_OrderLine VALUES (21,22,23,24,4,NULL,25,NULL)
 INSERT INTO C_OrderLine VALUES (26,27,28,29,4,15,NULL,13)
 INSERT INTO C_OrderLine VALUES (52,51,51,53,4,NULL,54,13)
 INSERT INTO C_OrderLine VALUES (61,60,60,62,4,15,NULL,13)
 INSERT INTO C_OrderLine VALUES (63,64,65,66,4,NULL,NULL,13)
 INSERT INTO M_Product VALUES (15,'N') // should only be in DB for mutants 2-3, 13-16, 18-20 (when j must = 1)
 INSERT INTO M_Product VALUES (15,'C')

```
INSERT INTO M_InOutLine VALUES (14,13)
INSERT INTO M_InOut VALUES (14,'IP') // should only be in DB for mutants 12, 20 (when i must =0)
INSERT INTO M_InOut VALUES (14,'EE')
INSERT INTO C_DocType VALUES (6,'SOO','CC')
INSERT INTO C_DocType VALUES (6,'SOO','ON')
INSERT INTO C_DocType VALUES (6,'XXX','DD')
INSERT INTO C_DocType VALUES (6,'XXX','OB')
```

Query 5

View query appears in: R_Request_v

Schema:

```
<schema>
  <table name="R_Request">
    <column name="IsActive" type="char"/>
    <column name="Processed" type="char"/>
    <column name="DateNextAction" type="date"/>
  </table>
</schema>
```

SQL:

```
SELECT * FROM R_Request WHERE IsActive='Y' AND Processed='N' AND sysdate > DateNextAction
```

WHERE clause as a Minimal DNF Predicate:

Letting $a=IsActive='Y'$, $b=Processed='N'$, $c=sysdate > DateNextAction$ the WHERE clause predicate in minimal DNF is abc

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
3 LOF mutants are generated as follows: bc, ac, ab

Test points needed to kill mutants in terms of literal values:

111 detects a FALSE mutant:	original true and mutant false
011 detects an LOF:	original false and mutant true
101 detects an LOF:	original false and mutant true
110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO R_Request VALUES ('Y','N',SYSDATE-1)
INSERT INTO R_Request VALUES ('A','N',SYSDATE-2)
INSERT INTO R_Request VALUES ('Y','B',SYSDATE-3)
INSERT INTO R_Request VALUES ('Y','N',SYSDATE+1)
```

Query 6

View query appears in: RV_BPartnerOpen

Schema:

```
<schema>
  <table name="C_Payment_v">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="IsActive" type="char"/>
    <column name="Created" type="date"/>
    <column name="CreatedBy" type="varchar"/>
    <column name="Updated" type="date"/>
    <column name="UpdatedBy" type="varchar"/>
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_Currency_ID" type="number"/>
    <column name="PayAmt" type="number"/>
    <column name="MultiplierAP" type="number"/>
    <column name="C_Payment_ID" type="number"/>
    <column name="C_InvoicePaySchedule_ID" type="number"/>
    <column name="DateTrx" type="date"/>
    <column name="IsAllocated" type="char"/>
    <column name="DocStatus" type="varchar" />
  </table>
</schema>
```

SQL:

```
SELECT p.AD_Client_ID, p.AD_Org_ID, p.IsActive, p.Created, p.CreatedBy, p.Updated, p.UpdatedBy,
p.C_BPartner_ID, p.C_Currency_ID, p.PayAmt*MultiplierAP*-1 AS Amt,
p.C_Payment_ID*p.MultiplierAP*-1 AS OpenAmt, p.DateTrx AS DateDoc, null FROM C_Payment_v p,
WHERE p.IsAllocated='N' AND p.C_BPartner_ID IS NOT NULL AND p.DocStatus IN ('CO','CL')
```

WHERE clause as a Minimal DNF Predicate:

Letting a=p.IsAllocated='N', b=p.C_BPartner_ID IS NOT NULL, c=p.DocStatus IN ('CO','CL') the WHERE clause predicate in minimal DNF is abc

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
3 LOF mutants are generated as follows: bc, ac, ab

Test points needed to kill mutants in terms of literal values:

111 detects a FALSE mutant:	original true and mutant false
011 detects an LOF:	original false and mutant true
101 detects an LOF:	original false and mutant true
110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO C_Payment_v VALUES
(1,2,'A',SYSDATE+1,'B',SYSDATE+2,'C',3,4,5,6,7,8,SYSDATE+3,'N','CO')
INSERT INTO C_Payment_v VALUES
(9,10,'D',SYSDATE+4,'E',SYSDATE+5,'F',11,12,13,14,15,16,SYSDATE+6,'G','CL')
INSERT INTO C_Payment_v VALUES
(17,18,'H',SYSDATE+7,'I',SYSDATE+8,'J',null,19,20,21,22,23,SYSDATE+9,'N','CO')
```

```
INSERT INTO C_Payment_v VALUES  
(24,25,'K',SYSDATE+10,"L",SYSDATE+11,"M",26,27,28,29,30,31,SYSDATE+12,'N',"AA")
```

Query 7

View query appears in: RV_OpenItem

Schema:

```
<schema>
  <table name="RV_C_Invoice">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="DocumentNo" type="number"/>
    <column name="DateInvoiced" type="date"/>
    <column name="C_Invoice_ID" type="number"/>
    <column name="DocStatus" type="varchar"/>
    <column name="IsSOTrx" type="char"/>
    <column name="isPaid" type="char"/>
    <column name="IsPayScheduleValid" type="char"/>
    <column name="GrandTotal" type="number"/>
    <column name="C_Currency_ID" type="number"/>
    <column name="C_ConversionType_ID" type="number"/>
    <column name="C_PaymentTerm_ID" type="number"/>
  </table>
  <table name="C_PaymentTerm">
    <column name="C_PaymentTerm_ID" type="number"/>
    <column name="NetDays" type="number"/>
    <column name="DiscountDays" type="number"/>
    <column name="Discount" type="number"/>
  </table>
</schema>
```

SQL:

```
SELECT i.AD_Org_ID, i.AD_Client_ID, i.DocumentNo, i.C_Invoice_ID, i.C_Order_ID,
i.C_BPartner_ID, i.IsSOTrx, i.DateInvoiced, p.NetDays, i.C_PaymentTerm_ID, i.DateInvoiced,
i.DateInvoiced + p.DiscountDays AS DiscountDate,
ROUND(i.GrandTotal*p.Discount/100,2) AS DiscountAmt, i.GrandTotal, i.C_Currency_ID,
i.C_ConversionType_ID,
i.C_PaymentTerm_ID, i.IsPayScheduleValid, null AS C_InvoicePaySchedule_ID FROM RV_C_Invoice I
INNER JOIN C_PaymentTerm p ON (i.C_PaymentTerm_ID=p.C_PaymentTerm_ID) WHERE
i.C_Invoice_ID <> 0 AND i.IsPayScheduleValid<>'Y' AND i.DocStatus<>'DR'
```

WHERE clause as a Minimal DNF Predicate:

Letting a=i.C_Invoice_ID <> 0, b=i.IsPayScheduleValid<>'Y', c=i.DocStatus<>'DR' the WHERE clause predicate in minimal DNF is abc

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
3 LOF mutants are generated as follows: bc, ac, ab

Test points needed to kill mutants in terms of literal values:

111 detects a FALSE mutant:	original true and mutant false
011 detects an LOF:	original false and mutant true
101 detects an LOF:	original false and mutant true
110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO RV_C_Invoice VALUES (1,2,3,4,5,SYSDATE+1,6,"A",'B','C','D',7,8,9,10)
INSERT INTO RV_C_Invoice VALUES (11,12,13,14,15,SYSDATE+2,0,"E",'F','G','H',16,17,18,10)
INSERT INTO RV_C_Invoice VALUES (20,21,22,23,24,SYSDATE+3,6,"I",'J','K','Y',25,26,27,10)
INSERT INTO RV_C_Invoice VALUES (29,30,31,32,33,SYSDATE+4,6,"DR",'L','M','N',34,35,36,10)
INSERT INTO C_PaymentTerm VALUES (10,39,40,41)
```


Query 8

View query appears in: RV_OpenItem

Schema:

```
<schema>
  <table name="RV_C_Invoice">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="C_BPartner_ID" type="number"/>
    <column name="C_Order_ID" type="number"/>
    <column name="DocumentNo" type="number"/>
    <column name="DateInvoiced" type="date"/>
    <column name="C_Invoice_ID" type="number"/>
    <column name="DocStatus" type="varchar"/>
    <column name="IsSOTrx" type="char"/>
    <column name="isPaid" type="char"/>
    <column name="IsPayScheduleValid" type="char"/>
    <column name="GrandTotal" type="number"/>
    <column name="C_Currency_ID" type="number"/>
    <column name="C_ConversionType_ID" type="number"/>
    <column name="C_PaymentTerm_ID" type="number"/>
  </table>
  <table name="C_InvoicePaySchedule">
    <column name="C_Invoice_ID" type="number"/>
    <column name="C_InvoicePaySchedule_ID" type="number"/>
    <column name="DueDate" type="date"/>
    <column name="DiscountDate" type="date"/>
    <column name="DiscountAmt" type="number"/>
    <column name="DueAmt" type="number"/>
    <column name="isValid" type="char" />
  </table>
</schema>
```

SQL:

```
SELECT i.AD_Org_ID, i.AD_Client_ID, i.DocumentNo, i.C_Invoice_ID, i.C_Order_ID,
i.C_BPartner_ID, i.IsSOTrx, i.DateInvoiced, ips.DueDate, ips.DiscountDate, ips.DiscountAmt,
ips.DueAmt AS GrandTotal, i.C_Currency_ID, i.C_ConversionType_ID, i.C_PaymentTerm_ID,
i.IsPayScheduleValid, ips.C_InvoicePaySchedule_ID FROM RV_C_Invoice i
INNER JOIN C_InvoicePaySchedule ips ON (i.C_Invoice_ID=ips.C_Invoice_ID) WHERE
ips.C_InvoicePaySchedule_ID <> 0
AND i.IsPayScheduleValid='Y' AND i.DocStatus<>'DR' AND ips.IsValid='Y'
```

WHERE clause as a Minimal DNF Predicate:

Letting a=ips.C_InvoicePaySchedule_ID <> 0, b=i.IsPayScheduleValid='Y', c=i.DocStatus<>'DR', d=ips.isValid='Y' the WHERE clause predicate in minimal DNF is abcd

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false

4 LOF mutants are generated as follows: bcd, acd, abd, abc

Test points needed to kill mutants in terms of literal values:

1111 detects a FALSE mutant: original true and mutant false

0111 detects an LOF:	original false and mutant true
1011 detects an LOF:	original false and mutant true
1101 detects an LOF:	original false and mutant true
1110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO RV_C_Invoice VALUES (1,2,3,4,5,SYSDATE+1,6,'A','B','C','Y',7,8,9,10)
INSERT INTO RV_C_Invoice VALUES (11,12,13,14,15,SYSDATE+2,6,'D','E','F','X',16,17,18,19)
INSERT INTO RV_C_Invoice VALUES (20,21,22,23,24,SYSDATE+3,6,'DR','H','T','Y',25,26,27,28)
INSERT INTO C_InvoicePaySchedule VALUES (6,38,SYSDATE+5,SYSDATE+6,39,40,'Y')
INSERT INTO C_InvoicePaySchedule VALUES (6,41,SYSDATE+7,SYSDATE+8,42,43,'Z')
INSERT INTO C_InvoicePaySchedule VALUES (6,0,SYSDATE+9,SYSDATE+10,44,45,'Y')
```

Query 9

View the query appears in: RV_WarehousePrice

Schema:

```
<schema>
  <table name="M_Product">
    <column name="M_Product_ID" type="number"/>
    <column name="C_UOM_ID" type="number"/>
    <column name="M_AttributeSet_ID" type="number"/>
    <column name="AD_Client_ID" type="number"/>
    <column name="IsSummary" type="char"/>
    <column name="IsActive" type="char"/>
    <column name="Discontinued" type="char"/>
    <column name="Value" type="varchar"/>
    <column name="Name" type="varchar"/>
    <column name="UPC" type="number"/>
    <column name="SKU" type="number"/>
  </table>
  <table name="M_ProductPrice">
    <column name="M_Product_ID" type="number"/>
    <column name="IsActive" type="char"/>
    <column name="Created" type="date"/>
    <column name="CreatedBy" type="varchar"/>
    <column name="Updated" type="date"/>
    <column name="UpdatedBy" type="varchar"/>
    <column name="M_PriceList_Version_ID" type="number"/>
  </table>
  <table name="C_UOM">
    <column name="C_UOM_ID" type="number"/>
    <column name="UOMSymbol" type="varchar"/>
  </table>
  <table name="M_AttributeSet">
    <column name="M_AttributeSet_ID" type="number"/>
    <column name="IsInstanceAttribute" type="char"/>
  </table>
  <table name="M_Warehouse">
    <column name="AD_Client_ID" type="number"/>
    <column name="AD_Org_ID" type="number"/>
    <column name="M_Warehouse_ID" type="number"/>
    <column name="Name" type="varchar"/>
    <column name="IsActive" type="char"/>
  </table>
</schema>
```

SQL:

```
SELECT w.AD_Client_ID, w.AD_Org_ID, CASE WHEN p.Discontinued='N' THEN 'Y' ELSE 'N' END
AS IsActive,
pr.Created, pr.CreatedBy, pr.Updated, pr.UpdatedBy, p.M_Product_ID, pr.M_PriceList_Version_ID,
w.M_Warehouse_ID, p.Value, p.Name, p.UPC, p.SKU, uom.C_UOM_ID, uom.UOMSymbol,
p.M_Product_ID, pr.M_PriceList_Version_ID, w.M_Warehouse_ID, w.Name AS WarehouseName,
COALESCE(pa.IsInstanceAttribute, 'N') AS IsInstanceAttribute
```

```
FROM M_Product p INNER JOIN M_ProductPrice pr ON (p.M_Product_ID=pr.M_Product_ID) INNER
JOIN C_UOM uom ON (p.C_UOM_ID=uom.C_UOM_ID) LEFT OUTER JOIN M_AttributeSet pa ON
(p.M_AttributeSet_ID = pa.M_AttributeSet_ID) INNER JOIN M_Warehouse w ON
(p.AD_Client_ID=w.AD_Client_ID) WHERE p.IsSummary='N' AND p.IsActive='Y' AND pr.IsActive='Y'
AND w.IsActive='Y'
```

WHERE clause as a Minimal DNF Predicate:

Letting a=p.IsSummary='N', b=p.IsActive='Y', c=pr.IsActive='Y', d=w.IsActive='Y' the WHERE clause predicate in minimal DNF is abcd

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false

4 LOF mutants are generated as follows: bcd, acd, abd, abc

Test points needed to kill mutants in terms of literal values:

1111 detects a FALSE mutant:	original true and mutant false
0111 detects an LOF:	original false and mutant true
1011 detects an LOF:	original false and mutant true
1101 detects an LOF:	original false and mutant true
1110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO M_Product VALUES (1,2,3,4,'N','Y','A',"AA","BB",5,6)
INSERT INTO M_Product VALUES (1,2,3,4,'C','Y','D',"CC","DD",7,8)
INSERT INTO M_Product VALUES (1,2,3,4,'N','E','F',"EE","FF",9,10)
INSERT INTO M_ProductPrice VALUES (1,'Y',SYSDATE+1,"",SYSDATE+2,"GG",11)
INSERT INTO M_ProductPrice VALUES (1,'G',SYSDATE+3,"H",SYSDATE+4,"I",12)
INSERT INTO C_UOM VALUES (2,"J")
INSERT INTO M_AttributeSet VALUES (3,'K')
INSERT INTO M_Warehouse VALUES (4,13,14,"HH",'Y')
INSERT INTO M_Warehouse VALUES (4,15,16,"II",'L')
```

Query 10

View the query appears in: C_Invoice_LineTax_vt

Schema:

```
<schema>
  <table name="C_InvoiceLine">
    <column name="AD_ORG_ID" type="number"/>
    <column name="C_Invoice_ID" type="number"/>
    <column name="C_InvoiceLine_ID" type="number"/>
    <column name="AD_Client_ID" type="number"/>
    <column name="IsActive" type="char" />
    <column name="Created" type="date"/>
    <column name="CreatedBy" type="varchar"/>
    <column name="Updated" type="date"/>
    <column name="UpdatedBy" type="varchar"/>
    <column name="C_UOM_ID" type="number" />
    <column name="Line" type="number" />
    <column name="Description" type="varchar" />
  </table>
  <table name="AD_Language">
    <column name="AD_Language" type="char"/>
    <column name="IsBaseLanguage" type="char"/>
    <column name="IsSystemLanguage" type="char"/>
  </table>
</schema>
```

SQL:

```
SELECT il.AD_Client_ID, il.AD_Org_ID, il.IsActive, il.Created, il.CreatedBy, il.Updated, il.UpdatedBy,
l.AD_Language, il.C_Invoice_ID, il.C_InvoiceLine_ID, null, null, null, null, il.Line, null, null, null,
il.Description, null, null, null, null, null, null, null, null, null, null, null, null, null, null FROM C_InvoiceLine il,
AD_Language l WHERE il.C_UOM_ID IS NULL AND l.IsBaseLanguage='N' AND
l.IsSystemLanguage='Y'
```

WHERE clause as a Minimal DNF Predicate:

Letting a=il.C_UOM_ID IS NULL, b=l.IsBaseLanguage='N', c=l.IsSystemLanguage='Y' the WHERE clause predicate in minimal DNF is abc

Mutants generated by the TRF-TIF tool:

1 FALSE mutant is generated as follows: false
3 LOF mutants are generated as follows: bc, ac, ab

Test points needed to kill mutants in terms of literal values:

111 detects a FALSE mutant:	original true and mutant false
011 detects an LOF:	original false and mutant true
101 detects an LOF:	original false and mutant true
110 detects an LOF:	original false and mutant true

Test points needed to kill mutants in terms of rows added to a test database:

```
INSERT INTO C_InvoiceLine VALUES
(1,2,3,4,'A',SYSDATE+1,'AA',SYSDATE+2,'BB',NULL,10,'EE')
INSERT INTO C_InvoiceLine VALUES (5,6,7,8,'B',SYSDATE+3,'CC',SYSDATE+4,'DD',9,11,'FF')
```

```
INSERT INTO AD_Language VALUES ('A','N','Y')  
INSERT INTO AD_Language VALUES ('B','C','Y')  
INSERT INTO AD_Language VALUES ('D','N','E')
```

References

References

- [1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Saywood. "Mutation Analysis," Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology. September 1979.
- [2] P. Ammann and J. Offutt. Introduction to Software Testing. Cambridge University Press. 2008, ISBN 0-52188-038-1.
- [3] M. Chan and S. Cheung. Testing Database Applications with SQL Semantics. *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*. Pages 363-374. Wollongong, Australia, March 1999, Springer-Verlag.
- [4] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, E. J. Weyuker. A Framework for Testing Database Applications. *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. Pages 147-157. Portland, OR. August 2000.
- [5] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, E. J. Weyuker. An AGENDA for Testing Relational Database Applications. *Software Testing, Verification and Reliability*, 14(1):17-44, March 2004. Wiley.
- [6] T. Y. Chen and M. F. Lau. An Empirical Study on the Effectiveness of the Greedy MUTP Criterion. *Proceedings of the 1998 International Conference on Software Engineering: Education and Practice*. Pages 338 – 344. Dunedin, New Zealand. January, 1998.
- [7] T. Y. Chen and M. F. Lau. Test Case Selection Strategies Based on Predicates. *Software Testing, Verification and Reliability*, 11(1):165-180, November 2001. Wiley.
- [8] T. Y. Chen, M. F. Lau and Y. T. Yu. MUMCUT: A Fault-Based Strategy for Testing Predicates. *Proceedings of the 6th Asia Pacific Software Engineering Conference*. Pages 606-613. Takamatsu, Japan. December, 1999.

- [9] J. J. Chilenski. An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion. Final Technical Report, DOT/FAA/AR-01/18, April 2001.
- [10] J. J. Chilenski and S. P. Miller. Applicability of Modified condition/decision coverage to Software Testing. *IEE/BCS Software Engineering Journal*, 9(5): 193-200, September 1994.
- [11] R. A. DeMillo, R.J. Lipton, F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34-41. April, 1978.
- [12] M. Emmi, R. Majumdar, K. Sen. Dynamic Test Input Generation of Database Applications. *Proceedings of the 2007 ACM International Symposium on Software Testing and Analysis*. Pages 151-162. London, United Kingdom. July, 2007.
- [13] W. G. J. Halfond and A. Orso. A Command-Form Coverage for Testing Database Applications. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Pages 69-80. Tokyo, Japan. September, 2006.
- [14] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 8(4):371-379, July 1982.
- [15] Rob Hierons, Mark Harman, and Sebastian Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability*, 9(4): 233-262, December 1999. Wiley.
- [16] K. S. How Tai Wah. A Theoretical Study of Fault Coupling. *Software Testing, Verification and Reliability*, 10(1):3-45, October, 2000. Wiley.
- [17] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*. Pages 249-258. Beijing, China. September, 2008.
- [18] G. Kaminski and P. Ammann. Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing. *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Pages 386-395. Denver, CO. April, 2009.
- [19] G. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection. *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Pages 167-176. Denver, CO. April, 2009.

- [20] G. Kaminski and P. Ammann. Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Double Fault Detection. *Proceedings of the Mutation Workshop at the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Denver, CO. April, 2009.
- [21] G. Kaminski and P. Ammann. Applying MCDC to Large DNF Expressions. *Proceedings of the 9th International Conference on Software Engineering Research and Practice*. Las Vegas, NV. July, 2010.
- [22] G. Kaminski and P. Ammann. Reducing Logic Test Set Size While Preserving Fault Detection. To appear in *Software Testing, Verification, and Reliability*. Wiley.
- [23] G. Kaminski, U. Praphamontipong, P. Ammann and J. Offutt. A Logic Mutation Approach to Selective Mutation Using Programs and Queries. Accepted with minor revisions by *Information and Software Technology, Special Issue on Mutation Testing*.
- [24] G. Kaminski, G. Williams and P. Ammann. Reconciling Perspectives of Logic Testing for Software. *Software Testing, Verification and Reliability*, 18(3):149-188, September 2008. Wiley.
- [25] G. M. Kapfhammer and M. L. Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. *Proceedings of the 9th European Software Engineering Conference*. Pages 98-107. Helsinki, Finland. September, 2003.
- [26] D. Richard Kuhn. Fault Classes and Error Detection Capability of Predicate Based Testing. *ACM Transactions on Software Engineering and Methodology*, 8(4): 411-424, October 1999.
- [27] M. F. Lau, Y. Liu and Y. T. Yu. On the Detection Conditions of Double Faults Related to Terms in Boolean Expressions. *Proceedings of the 30th Annual International Computer Software and Applications Conference*. Pages 403-410. Chicago, IL. September, 2006.
- [28] M. F. Lau, Y. Liu and Y. T. Yu. On the Detection Conditions of Double Faults Related to Literals in Boolean Expressions. *Proceedings of the 12th International Conference on Reliable Software Technologies*. Pages 55-68. Geneva, Switzerland. June, 2007.
- [29] M. F. Lau, Y. Liu and Y. T. Yu. Detecting Double Faults on Term and Literal in Boolean Expressions. *Proceedings of the 7th Annual International Conference on Quality Software*. Pages 117-126. Portland, OR. October, 2007.

- [30] M. F. Lau and Y. T. Yu. An Extended Fault Class Hierarchy for Predicate-Based Testing. *ACM Transactions on Software Engineering and Methodology*, 14(3): 247-276, July 2005.
- [31] Yu-Seung Ma, Jeff Offutt and Yong-Rae Kwon. MuJava : An Automated Class Mutation System. *Software Testing, Verification and Reliability*, 15(2):97-133, June 2005. Wiley.
- [32] Jeff Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions of Software Engineering and Methodology*, 1(1):3-18, January 1992.
- [33] J. Offutt and W. M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability*, 4(3):131-154, September 1994. Wiley.
- [34] J. Offutt and J. H. Hayes. A Semantic Model of Program Faults. *International Symposium on Software Testing and Analysis*. Pages 195-200. San Diego, CA. January 1996.
- [35] Jeff Offutt and Stephen Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337-334, May 1994.
- [36] Jeff Offutt and Jie Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165-192, September 1997. Wiley.
- [37] Jeff. Offutt, Gregg Rothermel and Christian Zapf. An Experimental Evaluation of Selective logic mutation. *Proceedings of the 15th International Conference on Software Engineering*. Pages 100-107, Baltimore, Maryland, May 1993.
- [38] V. Okun, P. Black and Y. Yesha. Comparison of Fault Classes in Specification-Based Testing. *Information & Software Technology*, 46(8): 525-533, June 2004.
- [39] M. Polo, M. Piattini, I. Garcia-Rodriguez. Decreasing the Cost of Mutation Testing with Second-order Mutants. *Software Testing, Verification and Reliability*, 19(2): 111-131, June 2009. Wiley.
- [40] M. J. Suarez-Cabal and J. Tuya. Using an SQL Coverage Measurement for Testing Database Applications. *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Pages 253-262. Newport Beach, CA. November, 2004.

- [41] Chang-ai Sun, Yunwei Dong, R. Lai, K. Y. Sim and T. Y. Chen. Analyzing and Extending MUMCUT for Fault-based Testing of General Boolean Expressions. *Proceedings of the 6th IEEE International Conference on Computer Information Technology*. Pages 184-189. Seoul, Korea. September, 2006.
- [42] J. Tuya, M. J. Suarez-Cabal, C. de la Riva. Full Predicate Coverage for Testing SQL Database Queries. Accepted for publication in *Software Testing, Verification and Reliability*. Wiley. Published online: Jan 15 2010, DOI: 10.1002/stvr.424.
- [43] J. Tuya, M. J. Suarez-Cabal, C. de la Riva. Mutating Database Queries. *Information and Software Technology*, 49(4): 398-417, 2007.
- [44] J. Tuya, M. J. Suarez-Cabal, C. de la Riva. SQLMutation: a Tool to Generate Mutants of SQL Database Queries. *Proceedings of the 2nd Workshop on Mutation Analysis*. Raleigh, NC. November, 2006.
- [45] S. A. Vilkomir and J. P. Bowen. Reinforced Condition / Decision Coverage (RC/DC): A new criterion for software testing. *Proceedings of the 2nd International Conference of Z and B Users*. Pages 295-313. Grenoble, France. January, 2002.
- [46] E. Weyuker, T. Goradia and A. Singh. Automatically Generating Test Data from a Predicate. *IEEE Transactions on Software Engineering*, 20(5): 353-363, May 1994.
- [47] D. Willmor and S. M. Embury. An Intensional Approach to the Specification of Test Cases for Database Applications. *Proceedings of the 28th ACM International Conference on Software Engineering*. Pages 102-111. Shanghai, China. May, 2006.
- [48] Y. T Yu and M. F. Lau. Comparing Several Coverage Criteria for Detecting Faults in Predicates. *Proceedings of the 4th International Conference on Quality Software*. Pages 14-21. Braunschweig, Germany. September, 2004.
- [49] Y. T Yu, M. F. Lau and T. Y. Chen. Using the Incremental Approach to Generate Test Sets: A Case Study. *Proceedings of the 3rd International Conference on Quality Software*. Pages 263-270. Dallas, TX. November, 2003.
- [50] Y. T Yu, M. F. Lau and T. Y. Chen. Automatic generation of test cases from Boolean specifications using the MUMCUT strategy. *Journal of Systems and Software*, 79(6): 820-840, June 2006.

CURRICULUM VITAE

Garrett K. Kaminski earned his B.A. in Psychology from the University of Virginia in 1993 and his M.S. in Information Systems from George Mason University in 1998. He has been employed as a software engineer in industry since 1997.